Peter Finzel

GRUNDLAGEN
6502-BEFEHLE
BEISPIELE

DAS ASSEMBLERBUCH

Einführung in die Assemblerprogrammierung der

ATARI®-COMPUTER

400 / 800 / 600 XL / 800 XL / 130 XE

DAS ASSEMBLERBUCH

für ATARI-Computer

Peter Finzel

Ich habe 2020 das Buch von Herrn Peter Finzel mit seiner vor Jahren gegebenen Erlaubnis nachgesetzt, damit wir im Gegensatz zum Einscannen eine kleinere und optisch bessere Datei haben. Dabei habe ich versucht möglichst dicht am Original zu bleiben.
Alle Fehlermeldungen deshalb an meine Mailadresse.
assemblerbuch@elephantxxl.de

Version 1.0

Peter Finzel Productions
Fürth/Bay.

1. AUFLAGE 1986

2. AUFLAGE 1987

Printed in Germany

Vorwort

Zugegeben - Bücher über die Maschinensprache des 6502-Prozessor gibt es schon einige. Wozu dann noch eines? Nun, dieses Buch erklärt Ihnen nicht nur allgemein den 6502-Prozessor, sondern ist ganz speziell für ATARI-Computer geschrieben. Das bedeutet, daß Sie Programm-Beispiele finden, die Sie sofort in Ihren Computer eingeben und ausprobieren können. Klar, dadurch lernt man nicht 'nur' Assembler, sondern bekommt auch gleich den praktischen Einsatz sowie einige wichtige Tips und Tricks vermittelt.

Der ATARI-Computer ist zweifellos eine Maschine, die sich hervorragend zur Programmierung in Assembler eignet. Seine großartigen graphischen und akustischen Fähigkeiten lassen sich in vollem Maße erst mit der Geschwindigkeit eines Maschinenprogrammes ausschöpfen. Natürlich erschöpft sich das Einsatzgebiet von Maschinenprogrammen nicht im spielerischen Gebiet. Auch gute Anwendungs-Programme, wie Textverarbeitungen oder Tabellenkalkulationen, sind meist in Assembler geschrieben, denn nur so wird flüssiger Programmablauf mit einem Minimum an Speicherplatzbedarf gekoppelt.

Eine Menge Gründe, sich mit der Maschinensprache Ihres Atari-Computers zu beschäftigen. Alles was Sie dazu brauchen, ist dieses Buch und einen leistungsfähigen Assembler. Als Grundlage dieses Buches dient der ATMAS-II Makroassembler. Nicht ohne Grund, denn er besitzt ein sehr gutes Preis-/Leistungsverhältnis, und ist vor allem mit deutscher Anleitung erhältlich.

Mein besonderer Dank gilt Petra Kolbe, die durch eifriges Korrekturlesen dem Fehlerteufel (hoffentlich!) den Garaus bereitet hat.

Fürth/Bay., September 1986 Peter Finzel

Inhaltsverzeichnis

Einleitung

Dieses Buch ist in drei Teile gegliedert, drei Teile, die Sie Schritt für Schritt in die Assemblerprogrammierung Ihres Atari-Computers einführen.

Teil I erläutert die Grundlagen. Sie lernen neue Zahlensysteme, den inneren Aufbau des 6502-Prozessors, die Grundzüge seiner Maschinensprache und die Werkzeuge zu deren Programmierung kennen.

Im zweiten Teil werden wir uns ausschließlich mit dem Befehlssatz des 6502 beschäftigen. Soweit es sinnvoll erschien, wurde zu jeder Gruppe von Befehlen ein praktisches Beispiel angefügt, das Sie direkt mit ATMAS-II eingeben können. Wie so oft, macht auch hier nur die Übung den Meister.

Der dritte Teil dient der Vertiefung des neu erworbenen Wissens durch etwas längere Beispielprogramme. Dabei wird jeweils ein speziell auf den Atari zugeschnittenes Thema wie Betriebssystem, Player-Missile Graphik oder Interrupts behandelt.

Sollten Sie auf Begriffe stoßen, die Ihnen nicht geläufig sind, so können Sie das 'Assembler-Vokabular' im Anhang E zu Rate ziehen. Dort finden Sie eine Zusammenstellung der wichtigsten Fachausdrücke.

Teil I: Grundlagen

1. Einführung in Assembler

In diesem einführenden Kapitel werden die grundlegenden Unterschiede von Assembler zu anderen Programmiersprachen gezeigt. Es ist anzunehmen, daß jeder Leser dieses Buches bereits Erfahrungen mit BASIC gesammelt hat, denn schließlich ist diese Sprache fest in jedem XL/XE-Computer eingebaut. Wir wollen daher besonders die wesentlichen Unterschiede zu BASIC herausarbeiten.
Natürlich ist es beim Schritt zur Maschinensprache nötig, daß man seine Denkweise etwas umstellt, da das 'Niveau' sehr viel niedriger anzusetzen ist. Es ist durchaus normal, daß Sie für einen einzigen BASIC-Befehl in Assembler ein längeres Unterprogramm benötigen.

1.1 Unterschied zu anderen Programmiersprachen

Der wesentlichste Unterschied, und das dürfte wohl einer der Gründe sein, weshalb Sie sich mit Assembler befassen wollen, ist die enorme Geschwindigkeit. Assemblerprogramme laufen 100, ja bis zu 1000 mal schneller als entsprechende BASIC-Programme. Und das kommt nicht von ungefähr: Assembler ist die Sprache, die der Mikroprozessor direkt versteht, während BASIC erst durch einen sogenannten 'Interpreter' für den Prozessor verständlich gemacht werden muß. Dieser Interpreter, der hauptsächlich aus einer Sammlung von Unterprogrammen (zu jedem BASIC-Befehl eines) besteht, ist selbstverständlich auch in Maschinensprache geschrieben, denn schließlich und endlich versteht der Prozessor nur diese eine Sprache.
Überhaupt, ganz egal in welcher Sprache Sie arbeiten, irgendwie muß diese Sprache auf die 'Maschinenebene' gebracht werden. Dies kann, wie bei BASIC, mittels eines Interpreters geschehen, der zu jedem Befehl ein Unterprogramm enthält. Schnellere Programme liefern schon sogenannte Compiler, die ein Programm von vorne herein in ein Maschinenprogramm umsetzen. Im Unterschied zu einem

Interpreter wird dabei die Zeit eingespart, die der Interpreter bei jedem Lauf eines Programmes zur Zuordnung der Unterprogramme zu den einzelnen Befehlen verbraucht.

Sie könnten jetzt einwenden, daß man sich ja gar nicht mit Maschinensprache auseinandersetzen muß, wenn ein Compiler das alles für Sie erledigen könnte. Falsch - denn auch die besten Compiler liefern einen bei weiten uneffektiveren Code als ein guter Assemblerprogrammier. Die schnellsten und dabei noch am wenigsten Speicherplatz benötigenden Programme müssen allemal in Assembler geschrieben werden.

Das Anfangs erwähnte 'Zurückschrauben' der Ansprüche beim Übergang von BASIC zu Maschinenprogrammen soll nun gleich an einem einfachen Beispiel konkret gezeigt werden. Nehmen wir an, es sollen zwei Zahlen (z. B. 17 und 4) addiert werden, das Ergebnis soll in der Variablen ERGEB festgehalten werden. In BASIC gibt es dabei kein Problem, man kann es eintippen wie man es in der Schule gelernt hat:

```
ERGEB = 17 + 4
```

Läßt man sich die Variable mit '?ERGEB' ausgeben, dann hat man das Resultat am Bildschirm stehen.

Versuchen wir das gleiche in Assembler. Die Einschränkung fängt schon damit an, daß es keinen 'Direkt-Modus' gibt. Wir haben ja schließlich kein BASIC-Programm eingegeben, sondern nur den (zur Fehlersuche recht praktischen) Direkt-Modus benutzt. Auch kein Problem - dann machen wir eben ein Assemblerprogramm daraus. Allerdings können Sie die nette Formelschreibweise von BASIC getrost vergessen. Für ein Assemblerprogramm müssen Sie die Formel in einzelne Rechenschritte zerlegen, ähnlich wie Sie das auf einem Taschenrechner machen würden. Stellen Sie sich dazu einen Taschenrechner vor, dessen (einziger) Speicher die Variable 'ERGEB' darstellen soll. Folgende Schritte wären zu tun:

1.) Zahl 17 in die Anzeige bringen
2.) Die Zahl 4 dazuaddieren
3.) Das Ergebnis im Speicher ablegen

Damit sind Sie schon ziemlich nahe an einem Assemblerprogramm. Nur hat der Prozessor natürlich keine Anzeige, sondern ein spezielles Register, den sogenannten Akkumulator. Und an Speicheplatz herrscht ja ebenfalls kein Mangel.

In Assembler würden die drei Schritte nun so lauten:

1.) Zahl 17 in den Akkumulator bringen
2.) Addiere die Zahl 4 zum Akku
3.) Speichere den Akku-Inhalt in der Speicherzelle ERGEB

Diesen Ablauf kann man direkt in ein kleines Assemblerprogramm umsetzen:

```
LDA #17
CLC
ADC #4
STA ERGEB
```

Keine Angst, wenn Ihnen diese Befehlskürzel noch recht seltsam anmuten, das alles wird in den nachfolgenden Kapiteln noch ausführlich erläutert. Soviel aber vorab: Mit LDA (Load Akkumulator) bringen wir die Zahl 17 in den Akku. Das 'CLC' sollten Sie im Moment noch übersehen, es ist nur Vorspiel zum nachfolgenden ADC-Befehl, der die Vier zum Inhalt des Akkus addiert. Das Ergebnis wird mit einem STA (Store Akkumulator) im Speicher abgelegt, vorausgesetzt, wir haben die Variable ERGEB vorher definiert. Damit sind wir bei einem weiteren Unterschied zu BASIC: Jeder verwendete Name muß auch irgendwo im Programm definiert sein.

Natürlich ist dieses Code-Stück noch lange kein lauffähiges Programm, aber Sie sehen jetzt schon, wo es lang geht.

1.2 Unterschiede zu BASIC

Während in BASIC auch mit Brüchen oder sehr großen Zahlen gearbeitet werden kann, rechnen die Befehle der Maschinensprache nur mit 'Bytes', die einen Zahlenbereich von 0-255 haben. Das heißt, hätten wir im obigen Beispiel die Zahlen 170 und 140 addieren wollen, dann wäre der Zahlenbereich bereits überschritten. Das ist eben der Preis, den man für Geschwindigkeit zahlen muß. Es ist deswegen aber nicht gesagt, daß man keine größeren Zahlen als 255 verarbeiten kann. Sicherlich kann man, nur wird das Assemblerprogramm aufwendiger (und langsamer...) da mehrere Befehle für eine einzige Rechnung nötig werden. Nehmen Sie nur die von BASIC benutzten Rechenroutinen (das FP-ROM), die einen Speicherplatz von über 2000 Bytes belegen!

Direkte Befehle zur Ein- und Ausgabe von Texten oder Zahlen gibt es in Assembler auch nicht, d.h. auf 'Komfort' wie 'PRINT' oder 'INPUT' muß man verzichten. Will man Text auf dem Bildschirm erscheinen lassen, dann muß man das 'Betriebssystem', ein über 10.000 Bytes langes Maschinenprogramm im ROM des Computers, bemühen. Dort sind eine Reihe von Unterprogrammen vorhanden, die es erlauben, ein Zeichen von der Tastatur zu holen oder ein Zeichen auf dem Bildschirm auszugeben.

Sie sehen, daß so manches, was in BASIC selbstverständlich erscheint in Assembler zu einem kleinen Problem werden kann. Dennoch ist Assembler viel flexibler als BASIC und erlaubt Ihnen, den Atari-Computer optimal auszunutzen.

2. Zahlensysteme

Ein besonders wichtiges Kapitel beim Schreiben von Assemblerprogrammen ist die Beherrschung von unterschiedlichen Zahlensystemen. Wir Menschen sind es ja gewohnt, im Zehnersystem zu rechnen, wogegen für die Assemblerprogrammierung das Zweier- und 16er-System interessant ist.

Was verbirgt sich aber eigentlich hinter dem Begriff 'Zehnersystem'? Nichts weiter, als daß jede Ziffer den zehnfachen Stellenwert der vorausgehenden hat. Nehmen wir als Beispiel die Zahl 5467. Man kann diese Zahl auch so ausdrücken:

$$5467 = 7x1 + 6x10 + 4x100 + 5x1000$$

Den niedrigsten Stellenwert hat die Sieben, gefolgt von der Sechs, die angibt, wie oft die Zehn enthalten ist. Die Vier zeigt, wie oft die nächste Potenz von zehn, die 100, benötigt wird etc. Man bezeichnet diese Anordnung als Zahlensystem zur Basis 10, weil sich die Stellenwerte (1, 10, 100, 1000...) jeweils um den Faktor 10 vergrößern.

1
10
100
1000
10000

Stellenwerte des l0er-Systemes

Natürlich könnte man auch jede andere Zahl mit Ausnahme der Null und der Eins als Basis eines Zahlensystemes verwenden. Bei den Menschen hat sich nur die Zehn als 'Standard' durchgesetzt, wahrscheinlich weil wir zehn Finger haben. Bei einem Computer sind die Verhältnisse dagegen ganz anders. Hier wird mit digitaler Logik gearbeitet, die nur zwei Zustände kennt: Entweder ist Spannung vorhanden, oder es liegt keine Spannung an. Diesen beiden Zuständen ordnet man Ziffern zu: Keine Spannung bedeutet '0', während das Vorhandensein von Spannung mit '1' bezeichnet wird.

2.1 Das Zweiersystem

Überlegen Sie sich einmal, wie ein Zahlensystem zur Basis 2 aussehen würde. Ausgehend vom uns bekannten Zehnersystem, in dem 10 Ziffern (0 - 9) benötigt werden, können wir sofort sagen, daß es in einem 'dualen' Zahlensystem nur zwei Ziffern, die Null und die Eins, geben wird. Na wunderbar, das paßt ja genau zur digitalen Logik. Natürlich werden die Zahlen auch insgesamt länger werden, da sich der Stellenwert nicht mehr verzehnfacht, sondert nur noch verdoppelt. Nehmen wir wieder ein konkretes Zahlenbeispiel: 1011. Genau wie oben können wir diese Zahl auch so schreiben:

1011 = 1x1 + 1x2 + 0x4 + 1x8

1
2
4
8
16
32
64
128

Stellenwerte des Zweier-Sytemes

Sie sehen, die Stellenwerte (1,2,4,8) werden jetzt um den Faktor zwei größer. Mit diesem Schema können Sie übrigens auch Zahlen von einem System ins andere umrechnen: wenn Sie die obige Rechenaufgabe lösen (1+2+8=11) dann wissen Sie, daß die binäre Zahl '1011' der dezimalen Zahl '11' entspricht.

Der 6502-Mikroprozessor in Ihnem Atari rechnet mit Zahlen, die jeweils acht solcher Binärstellen, auch Bit genannt, umfassen. Eine solche Einheit aus acht Bit wird 'Byte' genannt. Ein solches Byte könnte in binärer Darstellung z.B. so aussehen:

11010110

Als kleine Rechenaufgabe können Sie gleich versuchen, diese Zahl mit dem obigen Schema in dezimale Darstellung umzuwandeln. Keine Angst, das ist gar nicht schwer.

Hier eine Hilfestellung:

```
  1      1      0      1      0      1      1      0
1x128 + 1x64 + 0x32 + 1x16 + 0x8 + 1x4 + 1x2 + 0x1
```

Na, haben Sie's? Jawohl, die richtige Lösung ist 214.

2.2 Hexadezimalsystem

An dem letzten Beispiel konnten Sie auch sehen, daß die binäre Darstellung recht unhandlich ist, da man sich solche ellenlange Zahlen recht schwer merken kann. Man hilft sich, indem man jeweils vier binäre Ziffer zu einer hexadezimalen Ziffer zusammenfaßt. Auf diese Weise gewinnt man aus einer achtstelligen Binärzahl eine zweistellige Hexadezimalzahl. Zur Darstellung von 16 Zuständen reichen die Ziffern von 0 bis 9 jedoch nicht aus, deshalb nimmt man noch die Buchstaben von A bis F zur Hilfe. In der nachfolgenden Tabelle sind den Hex-Ziffern die entsprechenden Binärzahlen und Dezimalzahlen gegenübergestellt.

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

An dieser Stelle vielleicht noch ein Hinweis auf einige begriffliche Feinheiten: Das 16-er System müßte korrekterweise mit 'Sedezimalsystem' bezeichnet werden, aber im sprachlichen Umgang hat sich das Wort 'Hexadezimalsystem' oder einfach 'Hex' eingebürgert. Ähnlich ist die Situation beim Zweiersystem: Man spricht vom 'dualen' Zahlensystem, bezeichnet hingegen die einzelnen Ziffern als 'Binärstellen'. In diesem Buch

werden einheitlich die Bezeichnungen 'hexadezimal' für das 16-er und 'binär' für das Zweiersystem verwendet. Alle Zahlen in diesem Buch, denen kein Zeichen vorgestellt ist, sind als dezimale Zahlen aufzufassen. Hexadezimale Zahlen sind immer mit einem führenden Dollar-Zeichen ($) erkenntlich, binäre Zahlen beginnen immer mit einem Prozent-Zeichen (%).

$E456 = 14x4096 + 4x256 + 5x16 + 6x1

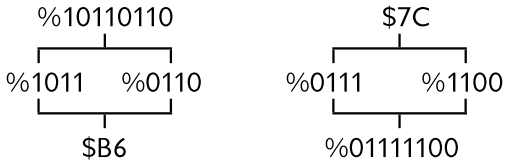
1
16
256
4096
65536

Stellenwerte im 16er System

Die Stellenwerte erhöhen sich jeweils (wie nicht anders zu erwarten) um den Faktor 16. Ein wichtige Frage bei diesen verschiedenen Zahlensystemen ist naturgemäß, wie man eines ins andere umrechnet.

2.3 Umrechnung

Am einfachsten gestaltet sich die Umwandlung vom Binär ins Hexadezimalsystem. Hier braucht man nur die Bits in Vierer-Gruppen zusammenzufassen und gemäß der obigen Tabelle umzusetzen. Ist die Stellenzahl nicht durch vier teilbar, setzt man einfach einige Nullen vor die Zahl. Genauso simpel kann man Hex-Zahlen in Binärzahlen zurückverwandeln, schließlich muss nur für jede Hex-Ziffer die entsprechende Binärschreibweise eingesetzt werden.



Die Umwandlung einer binären Zahl ins dezimale System haben wir praktisch schon besprochen: Man braucht nur diejenigen Stellenwerte zu addieren, die in der Zahl mit einer Eins besetzt sind. Gehen wir vom Beispiel links oben aus:

%10110110 : 2 + 4 + 16 + 32 + 128 = 182

Sehr ähnlich können wir auch bei Hex-Zahlen vorgehen. Da dort aber nicht nur zwei Zustände pro Ziffer erlaubt sind, muß der Wert jeder Ziffer (s. obige Tabelle) mit dem Stellenwert multipliziert werden. Das wird mit allen vorhandenen Ziffer gemacht und alle Produkte werden addiert.

$B6 : 11 x 16 + 6 = 182

$E456 : 14 x 4096 + 4 x 256 + 5 x 16 + 6 = 58454

Schwieriger wird es schon, wenn man dezimale Zahlen ins Zweier- oder 16er-System umrechnen will. Aber auch hier gibt es ein Schema, das ihnen nach kurzer Übung sicherlich nicht schwerfallen wird. Das geht so: Man teilt die dezimale Zahl durch die gewünschte Basis (in unserem Fall durch 2 oder 16) und bestimmt auch den Rest der Division. Das Ergebnis teilt man wieder durch die Basis, merkt sich den Rest usw. bis schließlich die Null als Ergebnis bleibt. Reiht man alle Reste aneinander (die erste Division lieferte die niedrigste Stelle) dann hat man die Darstellung im gewünschten Zahlensystem. Sehen Sie sich das gleich an einem Beispiel an:

```
182 : 2 =  91 Rest 0
 91 : 2 =  45 Rest 1
 45 : 2 =  22 Rest 1
 22 : 2 =  11 Rest 0
 11 : 2 =   5 Rest 1
  5 : 2 =   2 Rest 1
  2 : 2 =   1 Rest 0
  1 : 2 =   0 Rest 1
```

Ergebnis %10110110

Eigentlich recht einfach, oder? Es funktioniert natürlich genauso bei der Umwandlung einer dezimalen in eine hexadezimale Zahl:

```
58454 : 16 =  3653 Rest 6
 3653 : 16 =   228 Rest 5
  228 : 16 =    14 Rest 4
   14 : 16 =     0 Rest 14 ($E)
```

Ergebnis: $E456

Falls sich zweistellige Reste ergeben sollten, müssen diese in Hex-Ziffern umgesetzt werden. Nebenbei bemerkt: Das obige Ergebnis ist eine sehr

häufige Einsprungadresse ins Betriebssytsem des Atari-Computers.

Zweifellos ist der einfachste Weg zum Umrechnen von hexadezimalen Zahlen in Ihre dezimalen Darstellungen ein Taschenrechner, der eine eingebaute Funktion dazu hat. So ironisch das vielleicht klingen mag, aber es hat sich schon oft bewährt, wenn man so ein Gerät neben seinem Computer immer griffbereit liegen hat. Trotzdem muß man aber wissen, wie die Umrechnung funktioniert, denn es könnte ja sein, daß Sie einmal in die Verlegenheit kommen, z. B ein Programm zur Ausgabe von Hexzahlen schreiben zu müssen.

2.4 Binäres und hexadezimales Rechnen

Ein klein wenig sollten Sie mit diesen beiden im letzten Abschnitt vorgestellten Zahlensystemen auch rechnen können. Dazu genügt es prinzipiell, die Addition zu beherrschen, da man alle anderen Rechenarten davon ableiten kann.

Bei binären Additionen ist die Sache sehr einfach, da nie mehr als diese vier Fälle auftreten können:

```
0 + 0 =   0
0 + 1 =   1
1 + 0 =   1
1 + 1 =  10
```

Die letzte Zeile ist dabei zu lesen als: Eins plus eins gibt null Übertrag 1. Mit diesem Wissen können Sie schon alle binären Additionen ausführen. Beispiel:

```
  %10110010 (dez. 178)
 +%00110111 (dez.  55)
============================
  %11101001 (dez. 233)
```

Ein Übertrag wird dabei auf die nächste Stelle addiert.
Kleine Hilfestellung für das Rechnen mit Überträgen:

```
1 + 1 + 1=(1 + 1) + 1= 10 + 1= 11
```

Sie sehen, das Rechnen ist binär fast einfacher als im vertrauten Dezimalsystem. Etwas schwieriger gestaltet sich die Addition im Hexadezimalsystem, da es 16 verschiedene Ziffern gibt, deren Summen untereinander man sich merken müßte. Man kann sich aber leicht mit einer Eselsbrücke behelfen: Einfach die dezimalen Werte der Hexziffern addieren und das Ergebnis z.B. anhand Tabelle Abschnitt 2.2 zurückverwandeln. Sollte das Ergebnis größer als 15 sein, muß man 16 abziehen und bei der nächsten Stelle eine Eins als Übertrag addieren. Beispiel:

```
  $284E
 +$A32B
========
  $CE79
```

Hier könnte man sich bei den letzten beiden Ziffern so behelfen: 14 (für $E) + 11 (für $B) = 25, und das ist wiederum 9 plus 16. Ergebnis ist also die Neun und eins im Sinn für die nächste Stelle.

2.5 Negative Zahlen und Subtraktion

Die Subtraktion kann man von der Addition ableiten, wenn man über ein Bildungsgesetz für negative Zahlen verfügt. Hat man eine Differenz wie z. B. 8-3 vorliegen, müßte man nur zur 3 die entsprechende negative Zahl suchen und diese dann zur 8 addieren.

Zweierkompliment

Beim 6502 wird als Bildungsgesetz für negative Zahlen generell das sogenannte 'Zweierkomplement' verwendet. Es besagt, daß die fragliche Zahl zuerst invertiert und anschließend eine Eins dazu addiert wird. Invertieren bedeutet einfach, die Einsen gegen Nullen auszutauschen und umgekehrt. Ein Beispiel: Die binäre Darstellung der Zahl -10 ist bei einer Wortlänge von 8 Bit gesucht.

```
   %00001010  binär für 10

   %11110101  invertiert
+          1  eins dazu addieren
============
   %11110110  binär für -10
```

Der Übertrag, der beim Addieren der Eins in der höchsten Stelle auftritt, wird einfach ignoriert. Mit dieser Darstellung für die -10 kann man tatsächlich ganz normal rechnen. Als Beispiel berechnen wir 50-10, indem wir zu 50 -10 hinzuzählen:

```
   %00110010  binär für 50
+  %11110110  binär für -10
============
   %00101000  binär für 40
```

Der Übertrag wird wieder ignoriert. Es gibt ein paar Grundregeln, die man beim Rechnen mit negativen Zahlen beherzigen muß:

1.) Alle Rechenoperationen müssen mit der gleichen Wortlänge ausgeführt werden. Das gilt ganz besonders für die Bildung des Zweierkomplements. Hat man zu wenig Stellen, muß man VOR der Invertierung die vorderen Stellen mit Nullen auffüllen. Wenn man z.B. 8-stellige und 16-stellige Zahlen im Zweierkomplement addiert, ist das Ergebnis schlichtweg falsch!

2.) Der Zahlenbereich wird durch das Zweierkomplement verschoben. Bei Bytes hat man statt 0-255 nunmehr einen Bereich von -128 bis +127. Man beachte, daß diese beiden Darstellungen nur durch die Betrachtungsweise eines Bytes (oder eines Wortes) unterschieden werden. %11110110 kann genausogut -10 wie 246 bedeuten, je nachdem, ob man es als rein positives oder vorzeichenbehaftete Zahl ansieht.

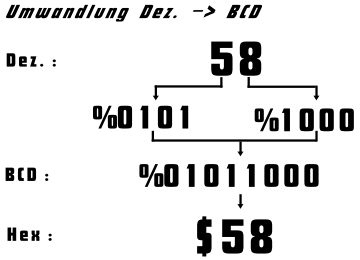
3.) Beim Zweierkomplement erkennt man eine negative Zahl daran, daß das höchstwertige Bit gesetzt ist. (Beispiel: siehe -10!). Positive Zahlen einschließlich der Null haben in diesem Bit eine Null. Außerdem ist der darstellbare Zahlenbereich nicht spiegelgleich: die mit 8-Bit kleinste mögliche Zahl ist 10000000 (-128 dez.), die größte dagegen %01111111 (+127 dez.)

Es empfiehlt sich daher dringend, vor dem Rechnen mit binären Zahlen festzulegen, ob man mit negativen Zahlen rechnen will, und welche Wortbreite (8 oder 16 Bit) verwendet wird.

2.6 BCD: Binär Codierte Dezimalziffern

Noch ein weiterer Code muß bei der Besprechung von Zahlensystemen erwähnt werden: der BCD-Code. Was so kompliziert klingt ist eine ganz einfache Sache. Mit vier Bit kann man bekanntlicherweise 16 verschiedene Zustände beschreiben. Zehn Zustände genügen bereits, um alle Ziffern des Zehnersystems darzustellen. Man definiert daher die Zustände %1010 bis %1111 als ungültig und hat so eine einfache Darstellung von Dezimalzahlen im Binärsystem. Immer zwei BCD-Ziffern finden in einem Byte Platz.

Dez.	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001



Besonders anschaulich wird die Sache im Hexadezimalsystem. Dort sind aufgrund der obigen Vorschrift die Buchstaben A-F als ungültig erklärt, d.h. nur noch 0-9 ist zulässig. Die 'Umrechnung' ist denkbar einfach:

87 (dez) -> in BCD %10000111 (oder $87)

Es genügt also, ein '$' vor die Zahl zu schreiben. Der Zahlenbereich eines Bytes geht jedoch beim BCD- Code von ursprünglich 0-255 auf 0-99 zurück. Dieser Code ist interessant, weil der 6502 sein Rechenwerk umschalten kann, so daß er anstatt mit normalen Binärzahlen mit BCD-Codes rechnen kann. Aufpassen muß man dabei beim überlauf: der tritt nämlich auf, wenn das Ergebnis größer als 99 ist. Mit diesem speziellen Modus des Prozessors arbeiten z.B. die Rechenroutinen von Atari-Basic.

3. Der 6502-Prozessor und sein Speicher

3.1 Modell des Prozessors

Anders als in höheren Programmiersprachen ist es beim Schreiben eines Assemblerprogrammes unerlässlich, den inneren Aufbau des Prozessors zu kennen. In höheren Sprachen wie PASCAL oder auch BASIC wird ja gerade versucht, die Programme möglichst unabhängig vom Prozessor zu gestalten, da man sie dann sehr einfach von einem Rechner auf einen anderen übernehmen könnte. In Assembler ist man jedoch streng an die Struktur des Prozessors gebunden, daher ist es auch nicht leicht möglich, z.B. ein 6502- Programm auf einen Z-80 Rechner zu übernehmen, da die beiden Mikroprozessoren recht unterschiedlich in ihren inneren Aufbau sind.

Das heißt natürlich nicht, daß man die Funktion jedes einzelnen der zigtausend Transistoren des 6502-Chips kennen müßte. Völlig ausreichend ist die Kenntnis eines Prozessor-Modelles, mit dem Sie sich im Bild 3.1-1 näher vertraut machen können.

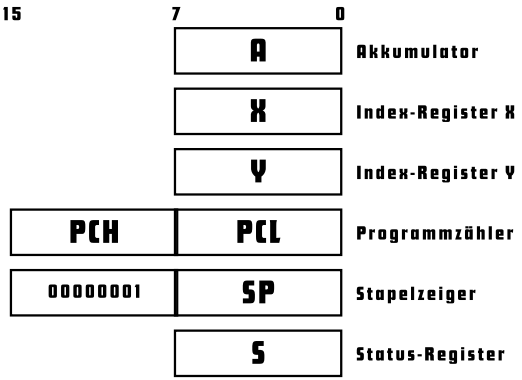


Bild 3.1-1: Modell des 6502-Prozessors

Die Kästen symbolisieren dabei spezielle Speicherzellen im Prozessor, die man für gewöhnlich Register nennt. Mit der Ausnahme eines einzigen, sind alle Register des 6502 acht Bit breit und können daher Zahlen im Bereich von 0 bis 255 aufnehmen. Das einzige 16-Bit (2 Byte) breite Register ist der Programmzähler (PC), den man als Zeiger auf den momentan ausgeführten Befehl verstehen kann. Wäre der PC auch nur 8 Bit breit, so könnte man nur 256 Bytes lange Programme adressieren (das wäre dann doch etwas wenig). Mit 16 Bit können (2 hoch 16) 65536 verschiedene Bytes adressiert werden, das entspricht genau den 64KByte des 800XL. Die Register im einzelnen:

Akkumulator (A, Akku):
Der Akku ist das wichtigste Register im 6502. Recht anschaulich läßt sich der Akku mit der Anzeige eines Taschenrechners vergleichen, denn alle Rechenoperationen müssen über den Akku abgewickelt werden.

X-Register (X):
Dient zur indizierten Adressierung von Speicherzellen. Darunter versteht man ein Verfahren zur Bearbeitung von Variablen-Feldern (Arrays) und Tabellen, das in späteren Kapiteln noch besprochen wird. Da der 6502 mit Registern nicht allzu reichlich versehen ist, wird dieses Register häufig auch als Allzweck-Register z. B. zur Aufnahme von Zwischenergebnissen benutzt.

Y-Register (Y)
Wie X ist Y ebenfalls ein weiteres Indexregister, das aber auch als Allzweckregister dient.

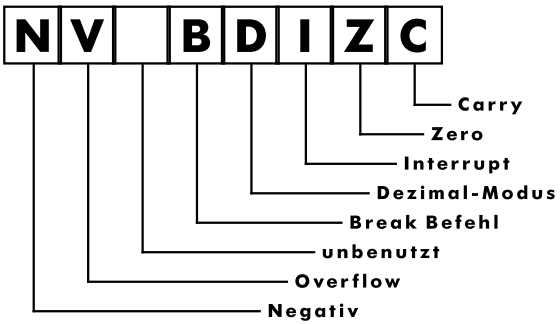
Programmzähler (PC, Program Counter)
Wie oben schon beschrieben, dient dieses Register als Zeiger auf den momentan bearbeiteten Befehl. Stellen Sie sich die Arbeitsweise des Prozessors so vor: Zuerst holt sich der 6502 das Byte aus dem Speicher, das durch den Programmzähler adressiert wird. Dieses Byte wird als Code eines Maschinenbefehles interpretiert. Erfordert der erkannte Befehl weitere Daten, so wird PC um eins erhöht und das nächste Byte gelesen. Nachdem der ganze Befehl gelesen und ausgeführt wurde, wird PC nochmal erhöht und der Ablauf beginnt von vorne, denn nun zeigt PC auf den Anfang des nächsten Befehles. Sprungbefehle (JMP) tun nichts anderes, als einen neuen Wert in PC zu schreiben, die Bearbeitung wird dann sofort an der neuen Adresse fortgesetzt. Als einziges Register verfügt der PC über 16 Bit, es hat also die doppelte Breite des Akkus.

Stapelzeiger (SP, Stackpointer)

Das SP-Register dient zur Bearbeitung des 'Stapels' (STACK), eines speziellen Speicherbereiches zur Verwaltung von Rücksprungadressen. Das kennen Sie sicher von BASIC her: Wenn ein Unterprogramm aufgerufen wird, dann muß beim unweigerlich folgenden RETURN noch bekannt sein, woher der Aufruf kam, denn dort muß das Programm schließlich fortgesetzt werden. Diese 'Rücksprungadressen' werden beim 6502 im Speicherbereich von $0100 bis $01FF aufbewahrt. Der Stapelzeiger verwaltet diesen Bereich, indem er auf den jeweils nächsten freien Platz zum Eintrag einer Adresse deutet. Daneben kann der Stack auch noch vom Programmierer verwendet werden, aber das werden Sie im Kapitel 4 noch genauer sehen.

Status-Register (P):

Jedes Bit dieses Registers kennzeichnet einen besonderen Zustand, z. B. wenn das Ergebnis der letzten Rechenoperation Null war oder ein Überlauf aufgetreten ist. Wie Sie später noch sehen werden, können die einzelnen Bits durch spezielle Verzweigungsbefehle abgefragt werden. Zwei Bits dieses Registers haben die Funktion eines Schalters: mit 'I' (s. unten) kann man eine gewisse Art von Interrupts nicht zulassen, mit 'D' kann das Rechenwerk von binärer Rechenweise auf dezimale umgeschalten werden.



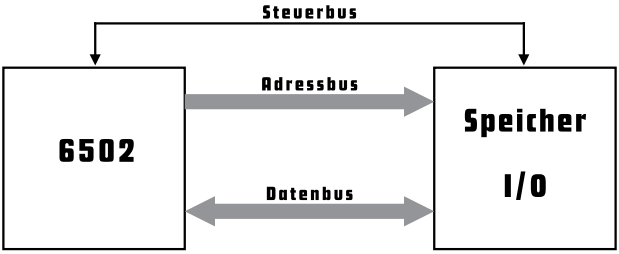
Hier die Funktionen der einzelnen Bits:

Bit 0 - C: (Carry) zeigt Überlauf ins 9. Bit an
Wichtig für Addition und Subtraktion.
Bit 1 - Z: (Zero) letztes Ergebnis war Null
Bit 2 - I : (Interrupt disable) verhindert die
Unterbrechung durch einen IRQ-Interrupt
Bit 3 - D: (Decimal Mode) schaltet das Rechenwerk
in den BCD-Modus um.
Bit 4 - B: (Break Command) zeigt an, daß ein BRK-
Befehl stattgefunden hat.
Bit 5 - (nicht belegt)
Bit 6 - V: (Overflow) Überlauf ins 8. Bit, dient
meist zum Rechnen mit Vorzeichen.
Bit 7 - N: (Negative) letztes Ergebnis negativ.

Keine Angst wenn gerade einiges vorgekommen ist, was Sie nicht verstanden haben. Das wird bei den entsprechenden Befehlen noch eingehend erläutert werden.

3.2 Der Speicher des Atari-Computers

Die wichtigste Vorgang bei der Arbeit des Prozessors ist der Zugriff auf den Speicher. Auf der Hardware-Seite wird der Zugriff mittels zweier sogenannter 'Bus'-Systeme ausgeführt. Unter 'Bus' versteht man hierbei nur eine Anzahl von parallelen Leitungen zwischen 6502-CPU und den angeschlossenen Speicherbausteinen. Jede dieser parallelen Leitungen kann ein Bit auf elektrischen Wege übertragen, d. h. wenn ein Byte übertragen werden muß, sind dazu acht parallele Leitungen nötig.



Wenn z.B. ein Byte in den Akku geladen werden soll, so legt der 6502 die Adressinformation auf den 'Adressen-Bus'. Der Speicher reagiert nun, indem er die Daten, die er unter der betreffenden Adresse gefunden hat, auf den 'Daten-Bus' legt. Die CPU liest die Daten vom Bus und bringt diese schließlich in den Akkumulator. Hardware-Profis werden einwenden, daß es eigentlich noch ein drittes Bus-Sytem gibt: den Steuer-Bus. Dort wird jedoch keine Information übertragen, sondern nur Hilfssignale gegeben, die z.B. anzeigen, ob Daten vom Speicher gelesen oder in den Speicher geschrieben werden sollen. Doch diese Details interessieren uns hier nur am Rande.

3.2.1 Maßeinheiten

Da der Programmzähler und die gesamte Logik zum Erzeugen der Adressen beim 6502 auf eine 'Breite' von 16 Bit ausgelegt ist, können maximal 2 hoch 16 gleich 65536 verschiedene Speicherzellen angesprochen werden. Der 'Daten-Bus' ist dagegen nur auf eine Breite von 8 Bit ausgelegt,

so daß unter jeder der 65536 Adressen ein 8-Bit breites Wort (ein Byte) zu finden ist.

> Man sagt daher, daß der 6502 eine Wortlänge von 8 Bit hat, er ist ein sog. '8-Bit Prozessor'. Auf diesem Sektor wurden von der Werbung schon viele halbe Wahrheiten verbreitet, grundsätzlich gilt: Besitzt ein Prozessor n Datenleitungen, so ist es ein n-Bit Prozessor.

Die Speichergröße gibt man in 'kByte' an, das sind Einheiten zu je 1024 Bytes. Der 6502 kann mit seinen 16 Adressleitungen maximal 64kByte direkt adressieren. Auf der anderen Seite wissen Sie aber sicher auch, daß ein 800XL schon allein 64kByte RAM sowie weitere 24kByte ROM enthält - ein Widerspruch? Nur scheinbar, denn es gibt Techniken, die Teile des Speichers umschaltbar machen, man spricht hier von 'Bank-Select'. Besonders intensiv wird von dieser Möglichkeit im 130XE mit seinen 128kByte Gebrauch gemacht.

Als weitere Einheit für die Speichermenge verwendet man beim 6502 noch den Begriff 'Page' (Seite). Man geht davon aus, daß der 16 Bit breite Programmzähler im Grunde aus zwei Teilen zu je acht Bit aufgebaut ist. Dies führt zu der recht bildlichen Vorstellung, daß der höherwertige Teil des Programmzählers jeweils die 'Speicherseite' bestimmt, der niederwertige Teil dagegen die Adresse innerhalb dieser Page angibt. Eine Page ist daher 256 Bytes lang und beginnt immer an einer Adresse, die im niederwertigen Teil Null ist (z.B. $7800). Den Sprung vom Ende einer Page zum Anfang der nächsten (z.B. von $77FF nach $7800) bezeichnet man als 'Page-Grenze'.

3.2.2 Arten des Speichers

Der gesamte Adressbereich ist beim Atari-Computer in drei Gruppen unterteilt: (s. auch Bild 3.2-2)

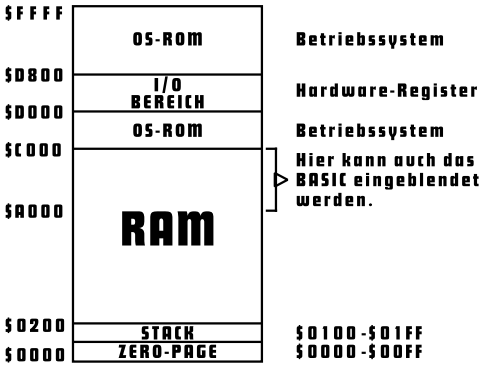


Bild 3.2-2: Grobe Speichermap des Atari-Computers

- RAM (Schreib-Lese Speicher):

Dies ist der größte Bereich, und dient zum Speichern von Programmen und Daten. Zwei besondere Bereiche sind die Zero-Page ('Null-Seite') in den Speicherzelle $0000 bis $00FF und die Page 1 von $0100 bis $01FF. Daten, die in der Zero-Page ablegt wurden kann der 6502-Prozessor besonders schnell verarbeiten, außerdem gibt es bestimmte Befehle, die nur auf die Zero-Page wirken. Die Page 1 dagegen dient als Stapelspeicher, d.h. hier werden unter anderem die Rücksprungadressen von Unterprogrammen verwaltet.

- ROM (Nur-Lese Speicher):

Wie der Name sagen, kann dieser Speicherbereich vom Prozessor nur gelesen, aber nicht modifiziert werden. In diesem Bereich ist das Betriebssystem (s. Kapitel III) und auch das BASIC untergebracht. Programme im ROM haben den Vorteil, daß sie gleich nach dem Einschalten des Computers vorhanden sind. Daher enthält das ROM auch Routinen zum Vorbereiten des Computers nach dem Einschalten und zum anschließenden Laden eines Programmes von Cassette oder Diskette. Bei den letzten genannten Fällen spricht man vom 'Boot-Vorgang', den dazu im ROM benötigten Code nennt man 'Boot-Loader'.

- I/O-Bereich:
Im eigentlichen Sinn handelt es sich hier nicht um 'Speicher', sondern nur um Möglichkeiten zur Eingabe und Ausgabe, die in Form von Speicherzellen angeordnet sind (der Fachmann spricht hier von 'Memory-mapped Input/Output'). Sehr anschaulich können Sie sich einzelne Bits als Schalter vorstellen, mit denen irgendwelche Zustände im Rechner oder an dessen Schnittstellen umgeschaltet werden können. Oder, im Falle der Eingabe, stellen Sie sich ein Bit als ein Lämpchen vor, an der die CPU einen Zustand ablesen kann. Diese einzelnen Bits sind jeweils in Achtergruppen angeordnet, so daß sie der 6502 wie eine Speicherzelle lesen kann. Oft sind dabei auch weniger als acht Elemente zusammengefaßt, so daß einige Bits einfach unbenutzt sind. Häufig werden Sie beim Atari auch I/O- Adressen finden, die beim Lesen und Schreiben ganz unterschiedliche Funktionen haben. Um beim obigen Vergleich zu bleiben, kann man beim Schreiben auf so eine Adresse die Schalter stellen, beim Lesen aber eine Reihe von Lämpchen abfragen. Die Speicherzellen des I/O-Bereiches nennt man 'Hardware-Register'. Der I/O-Bereich (beim ATARI von $D000 bis $D800) ist in kürzere Abschnitte unterteilt, die jeweils einem Chip zugeordnet sind. So belegt z.B. PIA den Bereich von $D300 bis $D303, dann folgt ANTIC von $D400 bis $D40F.

Eine interessante Tatsache ist auch, daß Maschinenprogramme und Daten vom Speicher nicht unterschieden werden. Läßt man den 6502 an einer beliebigen Stelle mit der Verarbeitung eines Programmes beginnen, so versucht er, den Inhalt des Speichers als Maschinenprogramm zu deuten. Befindet sich aber an dieser Stelle kein gültiges Programm, sondern etwa ein Zeichensatz, so erhält die CPU im günstigsten Fall einige unsinnige Befehle, wird aber früher oder später auf ungültige Befehlscodes stoßen. Das kann im schlechtesten Fall dazu führen, daß der Prozessor seine weitere Arbeit verweigert; der Computer 'stürzt ab'. Es liegt daher allein in der Verantwortung des Programmierers, daß sorgfältig zwischen Programm und Daten getrennt wird und somit nur gültige Maschinenprogramme ausgeführt werden.

4. Programmieren mit ATMAS-II

Wollte man ohne jegliche Hilfsmittel ein Programm in Assembler schreiben, so wäre das eine recht mühevolle Arbeit. Denn der Prozessor erwartet seine Befehle als Codes, als binäre Bit-Muster, die gewöhnlich durch zweistellige Hex-Zahlen dargestellt werden. So würde der Entwurf eines Programmes per Hand aussehen: Nachdem man sich überlegt hat, welche Befehle zum Einsatz kommen, müßten diese mit Hilfe einer Tabelle in entsprechende Hex-Codes umgewandelt werden. Diese Hex-Codes könnten dann nach nochmaliger Umwandlung in dezimale Zahlen z. B. mit POKE-Befehlen von BASIC in den Speicher gebracht werden und dann durch einen USR-Befehl aufgerufen werden.

Natürlich muß man auch die mit den Befehlen verknüpften Daten richtig berücksichtigen, achtgeben an welcher Adresse das Programm laufen soll, und zu guter letzt auch noch Sprungadressen und Entfernungen im Kopf behalten. Keine einfache Sache also, aber glücklicherweise haben Sie ja schließlich einen Computer, der uns solch lästige Aufgaben ersparen kann!

Das ist nun der Punkt, an dem ATMAS-II ins Spiel kommt. Dieses Programm erlaubt Ihnen, Assemblerprogramme in einer gut lesbaren Form einzugeben und erledigt sozusagen die 'Knochenarbeit', die nötig ist, um daraus ein für den Prozessor verständliches Programm zu machen. ATMAS-II ist daher im weitesten Sinne auch ein Art Compiler.

Noch einige begriffliche Feinheiten: Als 'Assembler' bezeichnet man sowohl die Programmiersprache als auch das zugehörige Übersetzungsprogramm. Ein Assembler erzeugt aus einem Quelltext, der beim ATMAS-II mit dem eingebauten Text-Editor eingegeben wird, das sogenannte Objektprogramm. Letzteres ist nichts anderes als ein lauffähiges Maschinenprogramm.

4.1 Eingabe der Befehle bei ATMAS-II

Als Voraussetzung zu diesem Buch sollten Sie die Anleitung des ATMAS-II Makroassemblers schon durchgelesen haben und zumindest wissen, wie der Bildschirm-Editor bedient wird. Das Format einer Eingabezeile lautet:

<Label> <Opcode> <Operand> <Kommentar>

Die Schreibweise in spitzen Klammern soll nur darauf hinweisen, daß. es sich nicht um die tatsächlichen Eingaben, sondern nur um Platzhalter für diese Eingaben handelt. <Label> ist eine Möglichkeit, eine Adresse mit einem Namen zu belegen, das ist nützlich für Sprungziele, oder zur Definition von Variablen und Konstanten. <Opcode> ist der Platzhalter für den eigentlichen Befehl, <Operand> gibt an, auf welche Daten sich der Befehl beziehen soll. Der <Kommentar> hat keinerlei Bedeutung für den Assembler, er ist nur Hilfsmittel für den Programmierer. Hier ein Beispiel für eine Zeile aus einem Assemblerprogrammes:

```
MARKE    LDA $D010    Joystick-Knopf gedrückt?
```

In dieser Zeile sind alle Komponenten vertreten. Natürlich braucht nicht jede Zeile vollständig sein. Wird in einer Zeile weder eine Variable definiert, noch wird die Zeile als Sprungziel gebraucht, so wäre es sinnlos, dort ein Label zu vergeben. Aus diesem Grund dürfen einzelne Komponenten solcher Zeilen auch fehlen. Sehen wir deswegen uns die einzelnen Zutaten einer Assemblerzeile näher an:

Labels

Sie dienen als Kennzeichnung einer bestimmten Stelle im Assemblerprogramm, anders ausgedrückt geben sie einer Speicherzelle einen symbolischen Namen. Denken Sie einmal kurz an BASIC: Wenn Sie einen GOTO-Befehl einsetzen wollen, müssen Sie immer die Zeilennummer des Sprungzieles kennen. Ganz ähnlich ist dies in Assembler, nur treten an Stelle der Zeilennummern die Speicheradressen der Assemblerbefehle. Da es recht mühsam wäre, sich alle Adressen eines längeren Assemblerprogrammes zu merken, hat man sich etwas einfacheres überlegt. Man schreibt einen Namen (Label) vor jede Zeile, zu der ein Sprung erfolgen soll. Später, wenn der Sprung dann programmiert werden muß, braucht man nur noch den Namen der Zeile anzugeben. Beim Assemblieren merkt sich der Assembler die Namen mit den zugehörigen

Adressen in der sog. Symboltabelle, und ersetzt schließlich die Labels im Programm durch Zahlenwerte. Versucht man, einigermaßen sinnvolle Namen zu vergeben, macht man sein Programm viel anschaulicher und transparenter. Sehen Sie sich als Veranschaulichung folgenden Auszug aus einem Programm an:

```
                ...
WARTEN  LDA RTCLOK+2
        CMP #100
        BEQ WEITER
        JMP WARTEN
*
WEITER  LDA ...
```

Auch wenn Sie die einzelnen Befehle noch nicht verstehen, können Sie die Struktur des Programmes schon allein durch die Namen der Labels erkennen. Zum Label 'WARTEN' wird verzweigt, wenn die interne 'Uhr' des Ataris noch nicht auf einem bestimmten Wert angelangt ist. Ist dieser erreicht, dann wird das Programm bei 'WEITER' fortgesetzt.

Zeilen, die nicht mit einem Label beginnen, müssen unbedingt eingerückt werden. Sollten Sie das Einrücken vergessen, interpretiert ATMAS-II den Opcode fälschlicherweise als Label und versucht, den Operanden als Opcode zu identifizieren, was natürlich mit einem 'ERROR' endet. Zum Einrücken genügt bereits ein Leerzeichen am Anfang der Zeile, man erhält aber eine besser Form des Listings, wenn man dazu einen Tabulator verwendet. Einzige Ausnahme dieser Regel: Kommentarzeilen (mit einem '*' in der ersten Zeile) dürfen nicht eingerückt werden.

Opcodes

Im Opcode-Feld steht nun das Kürzel (feiner ausgedrückt 'Mnemonic') des Maschinenbefehls. Die Mnemonics des 6502-Befehlscodes wurden so gewählt, daß sie genau drei Buchstaben umfassen, sie können daher schön untereinander geschrieben werden. Jede Zeile eines Assemblerprogrammes darf nur einen einzigen Befehl enthalten. Zu beachten ist auch, daß es neben den Befehlen des 6502-Prozessors noch einige 'Pseudo-Opcodes' gibt, die nur für den ATMAS-II Assembler bestimmt sind und nur organisatorische Aufgaben haben. Weiter unten wird unter 'Direktiven' ausführlicher auf diesen Punkt eingegangen.

Operanden

Der Operand gibt an, auf welche Adresse bzw. auf welchen Wert der Befehl wirkt. Dabei enthält der Operand auch einen Teil des Opcodes, da er durch die Adressierungsart angibt wie der Operand aufzufassen ist. Beispiel:

```
LDA #0
LDA 0
```

Die erste Anweisung besagt, daß die Zahl Null in den Akkumulator gebracht werden soll. Die zweite Anweisung bedeutet dagegen, daß der Inhalt der Speicherzelle Null in den Akku zu schaffen ist. Obwohl der Opcode gleich ist ergeben sich verschiedene Maschinenbefehle. Sie werden das im Abschnitt 'Adressierungsarten' noch ausführlicher sehen.

Kommentarfeld

Es wurde schon gesagt: Dieses Feld hat keinerlei Auswirkungen auf das erzeugte Maschinenprogramm. Anders als beim 'REM' in BASIC verbraucht Kommentar im Maschinenprogramm auch keinen Speicherplatz, da er beim Assemblieren nicht berücksichtigt wird. Einzig und allein wird der Text im Editor von ATMAS-II (das sog. Quellprogramm) länger. Nutzen Sie daher das Kommentarfeld reichlich, denn Assemblerprogramme tendieren dazu, unübersichtlich zu werden. Es ist auch erlaubt, ganze Zeilen als Kommentar zu verwenden, solche Zeilen müssen mit einem Asterisk ('*') beginnen. Beispiele:

```
* Diese Zeile ist Kommentar
        LDA #0   Akku loeschen
        ASL      ;Akku shiften
```

Bei ATMAS-II gilt noch eine Besonderheit, die man sich merken sollte. Kommentar nach Schiebe- und Rotationsbefehlen (s. Teil II) muß mit einem Strichpunkt beginnen da sonst die Gefahr besteht, daß eine falsche Adressierungsart gewählt wird.

4.2 Assemblerdirektiven

Neben den Befehlen des 6502-Prozessors beherrscht ATMAS-II noch einige 'Pseudo-Opcodes'. Diese zusätzlichen Anweisungen, man nennt sie auch häufig 'Assemblerdirektiven' geben dem Assembler zusätzliche Informationen, oder lassen es auch zu, beliebige Daten in den übersetzten Code (das sog. Objektprogramm) aufzunehmen. WICHTIG: Direktiven des Assemblers werden NICHT zu Maschinencodes übersetzt.

ORG

Mit der Direktive ORG wird die Anfangsadresse für das Objektprogramm festgelegt. Maschinenprogramme sind, soweit nicht spezielle Programmiertechniken angewendet werden, an feste Speicherbereiche gebunden. Soll ein Programm für eine andere Startadresse umgeschrieben werden, ändert man die ORG-Anweisung am Anfang des Programmes und assembliert neu. Ein Programm kann auch mehrere ORGs enthalten, man muß nur selbst darüber wachen, daß keine lebenswichtigen Speicherbereiche des Computers überschrieben werden bzw. keine Überlappungen auftreten.

```
ORG $A800
```

ATMAS-II bietet dem Programmierer einen geschützten Speicherbereich von $A800 bis ca. $BC00 (je nach Graphikstufe), in dem Objektprogramme abgelegt werden dürfen. Natürlich steht für kurze Programme auch die bekannte PAGE 6 ($0600 - $06FF) zur Verfügung. Programme für andere Adressen können erzeugt werden, wenn die Möglichkeit der getrennten logischen bzw. physikalischen Adresse (s. ATMAS-Handbuch, S.20) genutzt wird.

DFB, DFW und ASC

Diese drei Direktiven dienen zum Einfügen von beliebigen Daten in den erzeugten Objektcode. Nach DFB (steht für DeFine Byte) kann man eine Reihe von Byte-Daten angeben, die dann genauso im Objektcode erscheinen. Die Werte können dezimal oder hex angegeben werden. Beispiele:

```
DFB $20,32,$9B
DFB $80+37
```

```
HILFREG   DFB 128
```

Die letzte Zeile zeigt Ihnen, wie man in Assembler Variablen definiert und auch gleich vorbesetzt: HILFREG hat von Anfang an den Wert 128. Wenn Sie den Namen 'HILFREG' als Operand eines Befehles benutzen, wird die Adresse des mit DFB definierten Bytes eingesetzt. DFW wirkt ähnlich wie DFB, nur werden die nachfolgenden Daten als Worte angesehen. Worte bestehen aus zwei Bytes und werden beim 6502 in der Reihenfolge LSB, MSB gespeichert. Beispiel:

```
CHRSET    DFW $8000
```

Im Objektcode würde zuerst $00, dann $80 abgelegt. Soll das Wort später von einem Befehl angesprochen werden, so wird mit dem Label CHRSET nur das LSB adressiert. Das MSB spricht man in diesem Fall mit CHRSET+1 an. Die Direktive 'ASC' dient zum Ablegen ganzer Zeichenketten im Objektcode.

```
TEXT      ASC "Das ist ein Text"
```

Ebenfalls wird mit dem Label Text die Adresse des ersten Zeichens angegeben. ASC hat mehrere Arbeitsmodi, die im ATMAS-Handbuch (/5/ S. 22f) recht ausführlich erklärt sind.

EQU, EPZ

Diese beiden Pseudo-Opcodes dienen zur Definition von Konstanten. Prinzipiell unterscheidet sich eine Konstante nicht von einem Label, nur die Art der Definition ist anders. Während Labels immer den Wert des assemblerinternen Adresszählers bekommen, dürfen Konstanten jeden beliebigen Wert annehmen.

```
CIO       EQU $E456
GRUEN     EQU $A8
WAHR      EQU $FF
```

Mit anderen Worten: Anstatt Zahlen im Listing zu verwenden, kann man zuerst der Zahl einen symbolischen Namen zuordnen und diesen dann im Programm verwenden. Ist Ihnen schleierhaft, wozu dieser scheinbare Umweg gut sein soll? Es gibt mindestens zwei Gründe, die für die Verwendung von Konstanten sprechen. Erstens erklärt sich das Programm

damit selbst. Wenn Sie statt $A8 lieber die (natürlich zuvor definierte) Konstante GRUEN schreiben, dann ist jedem klar, daß dieser Wert für ein Farbregister bestimmt ist. Zweitens ist ein so geschriebenes Programm viel leichter veränderbar. Stellen Sie sich vor, sie hätten GRUEN öfter im Programm verwendet, stellen aber fest, daß die Farbe zu hell ist. Dann genügt es, die EQU-Definition am Anfang des Programmes zu ändern und das Programm neu zu assemblieren. Hätten Sie die Zahl direkt verwendet, müßten Sie mühevoll das ganze Programm danach durchsuchen. 'EPZ' ist nur eine Spezialform von EQU für Zero-Page Definitionen und braucht nicht verwendet zu werden (s. /5/ S.21).

Mit EQU definierte symbolische Namen eignen sich auch vorzüglich als Labels, die außerhalb des Programmes liegen. Hier zwei Beispiele:

```
CRSINH   EQU 752
CIOV     EQU $E456
```

In der ersten Zeile wird eine Variable definiert, die zweite Zeile definiert ein Sprungziel. Nur die spätere Verwendung mit einem Befehl entscheidet, ob es sich um eine Variable oder um ein Sprungziel (Label) handeln soll.

ATMAS-II besitzt neben den gerade angesprochenen Direktiven noch die OUT-Anweisung, die zur Erzeugung des Assemblerlistings dient. Außerdem gibt es noch Direktiven zur Definition von Makros, die im Teil III besprochen werden.

4.3 Adressrechnung

Nachdem im ersten Kapitel verkündet wurde, daß Rechenaufgaben für ein Assemblerprogramm mundgerecht in kleine Schritte aufgeteilt werden müssen, wird es Sie sicherlich erstaunen, wenn Sie folgende Zeilen in einem Assemblerlisting lesen würden:

```
LDA TAB/256+10
STA CHR*16+768
```

Schließlich sind hier die Rechenaufgaben in der gewohnten Formelschreibweise enthalten! Man muß aber beachten, daß Assemblerprogramme im Gegensatz zu BASIC vor der Verwendung assembliert werden müssen. Der Assembler hat bei diesem Vorgang auch die Güte, obige Rechenaufgaben gleich auszurechnen und die Ergebnisse als Operanden in das übersetzte Programm einzutragen. Man benutzt diese Fähigkeit des Assemblers gerne zur Berechnung von Adressen, etwa wenn man ein bestimmtes Element aus einer Tabelle haben möchte. Wichtig ist an dieser Stelle zu vermerken, daß Sie bei Assembler (und auch bei vielen Compiler-Sprachen) zwischen Rechenoperationen unterscheiden müssen, die beim Assemblieren erledigt werden und solchen, die erst beim Lauf des Programmes ausgeführt werden.

5. Ein erstes Beispiel

Nachdem Sie mit dem Innenleben des Prozessors und der Bedienung des Assemblers vertraut sind, können Sie sich an ein erstes kleines Assemblerprogramm wagen. Richtig bisher wurde noch kein einziger Maschinenbefehl besprochen. Trotzdem sollten Sie an dieser Stelle schon ein kleines Programm in Ihren Computer eingeben, sei es nur, um ein bißchen Gefühl für den Assembler zu bekommen. Danach fällt Ihnen auch das nächste (etwas trockentheoretische) Kapitel sicherlich leichter.

Wir wollen auch nur die einfachsten Formen der sogenannten Lade- und Speicherbefehle verwenden. Sie wissen ja sicher noch, daß beim 6502 alle Rechenoperationen über den Akkumulator laufen müssen. In diesem ersten Beispiel werden Sie sehen, wie eine bestimmte Zahl in den Akku gebracht wird, bzw. wie der Akku-Inhalt im Speicher abgelegt werden kann.

Gleichzeitig können Sie an dem folgenden kleinen Programm lernen, wie Sie mit den weiteren Beispielen in diesem Buch umgehen können. Am besten wäre es, wenn Sie diese Beispiele gleich beim Lesen mit ATMAS-II in Ihren Computer eingeben würden. Denn beim Programmieren in Assembler ist es wie bei vielen Dingen im Leben: Erst eigene Erfahrung macht den Meister.

Natürlich können wir mit den ersten Befehlen noch keine großen Sprünge machen, aber zur Nachahmung eines POKE-Befehles reicht es allemal. Damit auch auf dem Bildschirm etwas sichtbar wird, werden wir mit

```
POKE 712,10
```

den Rahmen des Textbildschirmes hellgrau einfärben.

5.1 POKE: Laden und Speichern

Zur Nachahmung des POKE-Befehles werden in Assembler gleich zwei Anweisung benötigt. Zuerst muß die Zahl 10 in den Akkumulator gebracht werden, dazu dient der 'LDA'-Befehl.

LDA steht für 'Load Accumulator', da der 6502-Mikroprozessor in den USA entwickelt wurde, beziehen sich alle Befehls-Kürzel auf die englischen Schreibweisen. Im zweiten Teil des POKE-Befehles muß der Inhalt des Akkus in die gewünschte Speicherzelle (in unserem Beispiel 712) übertragen werden, das ist die Aufgabe des STA-Befehles (Store Accumulator). Die Speicherzelle 712 ist übrigens das Schattenregister für die Hintergrundfarbe. Aus diesen beiden Befehlen können wir ein erstes Programm entwerfen:

```
*
* Simuliert POKE 712,10
*
      ORG $A800
      LDA #10
      STA 712
      BRK
```

Sie sehen schon, ein bißchen mehr als LDA und STA ist doch nötig. Die ersten drei Zeilen sind reiner Kommentar und könnten ebensogut entfallen. Sie sollten sich jedoch frühzeitig angewöhnen, Ihre Programme mit Überschriften und Kommentaren zu versehen, da besonders Assemblerprogramme dazu neigen, länger und unübersichtlich zu werden. Bei der ORG-Anweisung handelt es sich um eine Direktive für ATMAS-II, sie dient zur Festlegung der Anfangsadresse des Maschinenprogrammes. Wir benutzen im Beispiel den reservierten Bereich ab Adresse $A800. Jetzt kommen wir endlich zum LDA-Befehl, und Sie sehen, auch hier hat sich etwas merkwürdiges in Form eines Doppelkreuzes eingeschlichen. Dieses Symbol, man nennt es im 6502-Jargon das 'Immediate'-Symbol, kennzeichnet eine spezielle Adressierungsart, die Sie im nächsten Abschnitt noch näher kennenlernen werden. Sie bewirkt, daß die Zahl nach dem LDA-Befehl nicht als Adresse, sondern tatsächlich als Wert (man spricht hier auch von 'Datum') aufgefaßt wird. Das bedeutet, wenn Sie LDA 10 schreiben, dann würde der Inhalt der Adresse 10 in den Akku geladen, bei LDA #10 wird dagegen eine 10 in den Akku geschrieben. In der nächsten Zeile folgt der STA-Befehl, der den Inhalt des Akkus in der gewünschten Speicherzelle (in unserem Fall im Schattenregister der Hintergrundfarbe) ablegt. Auch der STA-Befehl kennt einige verschiedene Adressierungsarten, die ebenfalls im nächsten Abschnitt besprochen werden. Eines ist jedenfalls klar: Eine 'Immediate'-Adressierung kann es für STA nicht geben. Wissen Sie warum?

Break-Point

Damit sind wir bei der letzten Zeile angelangt. Hinter 'BRK' verbirgt sich der BREAK-Befehl, den Sie nicht mit der entsprechenden Taste Ihres Computers verwechseln dürfen. Dieser Befehl löst, obwohl er so kurz und unscheinbar ist, eine ganz Menge an Aktivitäten aus. Da Sie an dieser Stelle des Buches wahrscheinlich mit dem Begriff 'Software-Interrupt' noch nichts anfangen können, soll auch in diesem Fall die genaue Beschreibung auf ein späteres Kapitel verschoben werden. Merken Sie sich nur, daß Sie mit 'BRK' zurück zum Monitor des ATMAS-II kommen, und obendrein die Inhalte der Register des 6502-Prozessors angezeigt werden. Der Profi spricht hier von einem 'Break-Point'.

Nachdem Sie das Programm hoffentlich schon mit dem Editor des ATMAS-II Assemblers eingegeben haben, wollen wir es gleich laufen lassen. Zu diesem Zweck muß es natürlich erst einmal assembliert werden, das geht bei ATMAS-II mit der Tastenkombination <Control>-Y. Ein so kurzes Programm ist für ATMAS ein Kinderspiel, er meldet sich praktisch mit dem Tastendruck als fertig zurück. Er sagt Ihnen auch noch, wo das erzeugte Programm endet:

PHYSICAL ENDADDRESS: $A806

Sollten Sie statt dessen eine Fehlermeldung zu sehen bekommen, dann haben Sie sich irgendwo vertippt. Dann eine beliebige Taste drücken und das Programm gründlich mit dem Listing vergleichen. Benutzen Sie auch die Hinweise im ATMAS-Handbuch (/5/ Anhang B)!

3.2.2 Debugging mit dem Monitor

Haben Sie alle Tippfehler beseitigt, dann geht es ans Testen des Programmes. Der Profi spricht hier von 'Debugging' (zu deutsch etwa 'entwanzen'), Sie werden dabei vom ATMAS-Monitor tatkräftig unterstützt. Wenn Sie sich noch im Assembler-Modus befinden, dann drücken Sie bitte eine Taste um in den Editor zu gelangen. Nun noch <Control>-P eingeben, und der Monitor meldet sich zu Wort.

Damit Sie einmal sehen, was der Assembler erzeugt hat,

lassen wir uns den Maschinencode (das Objektprogramm) in hexadezimaler Form anzeigen. Drücken Sie dazu die Taste 'M' und geben dann die Adressen 'A800' und 'A806' ein. Auf die restlichen Fragen antworten Sie nur mit <Return>, dann müßten Sie folgende Ausgabe bekommen:

```
A800        A9 0A 8D C8 02 00
```

Das ist das erzeugte Maschinenprogramm (jedenfalls der Objekt-Code davon). Es ist noch anzumerken, daß im Monitor alle Ein- und Ausgaben hexadezimal sind. $A9 ist der Code für LDA #, $0A ist die hexadezimale Form von 10. Danach folgt der Code für STA in absoluter Adressierung ($8D). Bleibt nur noch der Operand des STA-Befehles: Man kommt auf die obige Darstellung, wenn man 712 ins Hexadezimalsystem umrechnet (Ergebnis: $02C8).

Man sollte wissen, daß die vorderen beiden Ziffern einer Adresse als das 'MSB' (Most Significant Byte) und die letzten beiden Ziffern als das LSB (Least Significant Byte) bezeichnet werden. Grundsätzlich gilt beim 6502 die Regel, daß Adressen in der Reihenfolge erst LSB, dann MSB abgespeichert werden. Damit haben wir auch den Hintergrund der nächsten beiden Hex-Zahlen $C8 und $02 geklärt. Den Abschluß bildet die Null, und die ist nichts anderes als der Code des BRK-Befehles.

Disassembler

Eine weitere hilfreiche Möglichkeit der Darstellung bietet der Disassembler ('Rückübersetzer'). Geben Sie dazu 'D' ein, gefolgt von der Adresse 'A800'. Danach 'RETURN' drücken, und schon erscheinen die Befehle wieder in besser lesbarer Form. Sie bemerken an dieser Stelle auch, daß das Disassembler-Listing nur noch bedingte Ähnlichkeit mit dem ursprünglichen Quelltext hat. Es fehlen sämtliche Kommentare, hätten wir Labels verwendet, so wären diese auch verschwunden. Der Objektcode (das übersetzte Programm) enthält alle diese Hilfsinformationen nicht mehr, daher kann der Disassembler sie auch nicht anzeigen. Mit 'X' wird der Disassembler wieder verlassen.

Jetzt ist es so weit: Wir starten den ersten Probelauf. Geben Sie dazu 'G' (für 'GO') ein und tippen Sie die Startadresse des Programmes 'A800' ein. Nun müßte die Bild-Umrahmung hell eingefärbt sein, außerdem sollte der Bildschirm die Inhalte der Register anzeigen:

```
  PC   AC XR YR SP NV.BDIZC
A805 0A 00 A8 F8 00110001
MONITOR
```

Diese Meldung sagt Ihnen, daß bei der Adresse $A805 (im PC) ein BRK-Befehl angetroffen wurde, und zeigt die Inhalte der Register und der Flags zu diesem Zeitpunkt. Sie sehen: im Akku steht noch die $0A vom LDA-Befehl. Die Werte für X-, Y-Register können bei ihnen auch anders sein, sie wurden vom Programm ja nicht benutzt. Unter 'SP' steht der Inhalt des Stapelzeigers, danach folgen alle Flags. Mit derselben Vorgehensweise können Sie auch alle Beispiele im Teil II dieses Buches starten und testen.

Der BRK-Befehl eignet sich auch hervorragend zur Fehlersuche in Maschinenprogrammen. Man setzt einfach ein BRK an die vermutlich fehlerhafte Stelle und läßt das Programm laufen. Sobald der BRK erreicht ist, wird der Monitor aktiv und zeigt die Registerinhalte an.

WICHTIG: Anders als bei einem 'RUN' in BASIC müssen Sie bei einem 'GO' auf ein Maschinenprogrammen immer eine Startadresse angeben!

6. Adressierungsarten

Die Mehrzahl der Assemblerbefehle benötigen zusätzliche Informationen, die angeben, welche Speicherzelle bzw. welcher Wert verarbeitet werden soll. Wie Sie im letzten Abschitt gesehen haben, ist ein Assemblerbefehl daher in zwei Teile gegliedert: Opcode und Operand. Der Opcode bestimmt die Funktion des Befehles. Der Operand gibt, falls benötigt, das Ziel oder die Quelle der Informationen an. Es kann sich im einfachsten Fall um Konstanten oder Adressen handeln, aber auch kompliziertere Ausdrücke sind zulässig. Sehen wir uns dazu ein Beispiel mit dem LDA-Befehl an:

```
LDA #237
LDA 1536
LDA 1536,X
LDA (88),Y
```

Alle vier Fälle benutzen den gleichen Opcode, der veranlaßt, daß ein Byte in den Akku geschrieben wird. Jedoch sind die Quellen dieses Bytes unterschiedlich, man sagt dann, daß der Befehl in verschiedenen 'Adressierungsarten' vorliegt. Optisch unterscheiden sie sich in verschiedenen Schreibweisen des Operanden, etwa mit vorangestelltem Doppelkreuz oder angehängtem ',X'. In diesem Abschnitt werden wir alle Adressierungsarten des 6502-Prozessors vorstellen und erläutern, wozu man sie verwenden kann.

6.1 Einfache Adressierungsarten

Immediate:
Häufig muß ein Register nur mit einer Konstante vorbelegt, oder eine Rechenoperation mit einer festen Zahl durchgeführt werden. Zu diesem Zweck dient die 'Immediate'- Adressierung (zu deutsch etwa: unmittelbare Adressierung), durch die der Operand nicht als Adresse sondern als Konstante betrachtet wird. Sie kennen diese Adressierungsart ja sicherlich noch aus dem letzten Abschnitt. Optisches Kennzeichen ist das Doppelkreuz vor dem Operanden, Zahlenwerte dürfen nur im Bereich von 0 bis 255 (dez.) liegen, da die Register ja nur 8 Bit umfassen. Die Befehle sind somit zwei Bytes lang (Opcode und Konstante). Beispiel:

```
LDA #237
LDA #MARKE/256
```

Das erste Beispiel würde die Zahl 237 in den Akku bringen.

Absolut:
Weit häufiger muß man eine bestimmte Speicherzelle ansprechen, um deren Inhalt zu laden, oder eine Zahl darin aufzubewahren. Zu diesem Zweck dient die absolute Adressierung, hier wird als Operand eine Speicheradresse angegeben. Da der 6502 sechzehn Adressleitungen hat, braucht man zur Darstellung einer absoluten Adresse zwei Byte, d.h. Befehle mit absoluter Adressierung sind 3 Bytes lang: Opcode - niederwertiger Adress-Teil (LSB) - höher- wertiger Adress-Teil (MSB). In der Schreibweise gibt es keine Besonderheiten, da die absolute Adressierung sozusagen der 'Normalfall' ist. Beispiele:

```
STA 712
LDA XPOS
```

Zero-Page:
Auch hier wird eine Adresse als Operand angegeben. Nur ist der Adressbereich auf die Speicherzellen 0 bis 255 (eben jene Zero-Page) beschränkt. Der Vorteil ist, daß solche Befehle weniger Speicherplatz als absolut adressierte Befehle benötigen (nur 2 statt 3 Bytes) und auch schneller ausgeführt werden. Im Grunde ist die Zero-Page Adressierungsart nur eine Spezialform der absoluten Adressierung, die dem Programmierer erlaubt, recht kompakte und schnelle Programme zu schreiben. Beispiele:

```
STA $C0
LDA SAVMSC
```

Ein guter Assembler braucht zur Unterscheidung von absoluter und Zero-Page Adressierung kein spezielles Zeichen, sondern erkennt das einfach an der Adressangabe.
Nehmen Sie daher wie im obigen Beispiel eine Zahl kleiner als 256 (oder einen Label, der für so eine Zahl steht), dann wird der Assembler automatisch die speicherplatzsparende Zero-Page Adressierung wählen.

Absolut Indiziert
Was tun Sie in BASIC, wenn Sie auf mehrere nacheinander liegende Adressen (z.B. ab 1536) zugreifen wollen? Richtig, Sie schreiben z. B. A=PEEK(1536+I) und verändern die

Variable I. Genauso funktioniert's auch in Assembler, nur gibt es dafür eine spezielle Adressierungsart namens 'Indiziert'. Die Rolle der Variablen I übernimmt ein Index-Register, die Adresse des Speicherzugriffes wird dann aus der angegebenen Adresse plus dem Inhalt des Index-Registers gebildet. Dem obigen PEEK-Befehl entspricht in Assembler die Anweisung

```
LDA 1536,X
```

Gegenüber der absoluten Adressierung unterscheidet sich die Schreibweise durch das an den Operanden mit Komma angehängte Index-Register. Nicht alle Befehle lassen allerdings die Wahl zwischen X und Y frei, bei manchen ist eines davon fest vorgeschrieben. Die im Operanden angegebene Adresse bezeichnet man als 'Basis-Adresse', den Inhalt des Index-Registers als 'Offset'. Da die Index-Register nur ein Byte enthalten, kann dieser Offset maximal nur 255 betragen.

Zero-Page Indiziert

Wiederum eine spezielle Form der absolut-indizierten Adressierung, die speziell auf die Zero-Page zugeschnitten ist. Wieder ist der Befehl um ein Byte kürzer und daher auch schneller, aber der Aktionsbereich ist auf den Speicherbereich von $0000 bis $00FF beschränkt. Beispiele:

```
LDA $80,X
STA $30,Y
```

Der Assembler unterscheidet diese Adressierungsart
selbständig von der absoluten Adressierung. Eine spezielle Situation bei dieser Adressierungsart tritt auf, wenn die Summe der Zero-Page Adresse und der Inhalt des Index-Registers mehr als $00FF ergeben, in diesem Fall findet ein sogenannter Wrap-Around (Umfaltung) statt. Nehmen wir als Beispiel LDA $F0,X und nehmen weiterhin im X-Register den Wert $20 an. Als tatsächliche Adresse (Operand plus Offset) ergibt sich die Adresse $0110, da aber nur die Zero-Page adressiert werden kann, wird das MSB auf Null gesetzt. Somit wird auf die Adresse $0010 zugegriffen.

6.2 Indirekte Adressierungsarten

Hier geht man davon aus, daß zwei nacheinander liegende Speicherzellen eine neue Speicheradresse enthalten. Gerne spricht man in so einem Fall auch von einem 'Zeiger' (oder Pointer, Vektor), der auf das gesuchte Byte deutet. Indirekte Adressierung brauchen Sie z. B. , wenn Sie ein Zeichen des Bildschirms per PEEK-Befehl lesen wollen. Die Anfangsadresse des Bildschirmspeichers ist beim Atari nicht fest vorgegeben, sondern steht in den Speicherzellen 88 und 89. Der indirekte Zugriff würde in BASIC so aussehen:

```
A=PEEK(PEEK(88)+PEEK(89)*256)
```

Damit würden wir indirekt, d.h. über den Umweg des Zeiger: in 88,89 auf das erste Zeichen des Bildschirmes zugreifen.

Indirekt-Indiziert
In Assembler geht das viel einfacher: Der Befehl

```
LDA (88),Y
```

erledigt die gleiche Aufgabe wie die verschachtelten PEEKs Aber was hat das 'Y' am Ende zu suchen? Das ist eine Eigenheit des 6502-Prozessors, denn dieser kennt keine einfache indirekte Adressierung. Nein, der 6502 kann nur indirekt zugreifen, wenn er gleichzeitig noch indizieren, also das Index-Register zur Adresse zählen darf. In Beispiel würde er die Bildschirmadresse aus 88,89 holen, dazu das Y-Register addieren und auf diese Adresse schließlich zugreifen. Wenn wir reine indirekte Adressierung wünschen, muß das Y-Register (und nur dieses ist hier erlaubt) zuvor auf Null gesetzt werden. Vielleicht wundern Sie sich jetzt, warum man das so kompliziert gemacht hat, aber Sie werden bald merken, daß es in vielen Fällen sehr praktisch ist.

Da diese Adressierung wirklich sehr wichtig zum Verständnis des 6502-Assemblers ist, soll sie an einem Beispiel nochmals erklärt werden. Sehen Sie sich dazu das Bild 6-1 an.

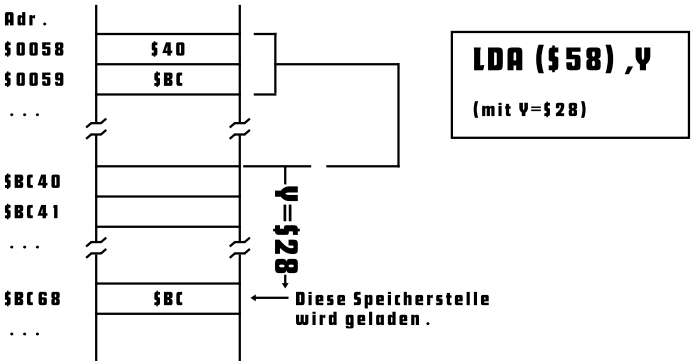


BILD 6-1: Indirekt-Indiziert

Aus den Inhalten der Speicherzellen 88 und 89 ergibt sich die Adresse $BC40, dazu wird der Inhalt des Y-Registers ($28) addiert, im Endeffekt wird der Inhalt der Adresse $8C68 in den Akku gebracht. Wichtig: Der indirekte Zeiger (d.h. die Adresse im Operanden) muß in der Zero-Page liegen.

Indiziert-Indirekt

Im Unterschied zum obigen ist bei dieser Adressierungsart die Reihenfolge des Ablaufes umgekehrt: Zuerst wird indiziert und dann erst indirekt zugegriffen. Das schlägt sich auch auf die Schreibweise nieder:

```
LDA ($C0,X)
STA  (SAVMSC,X)
```

Als Operanden sind wiederum nur Zero-Page Adressen zulässig, als Index-Register darf diesmal nur das X-Register verwendet werden. Den Mechanismus dieser Adressierungsart können Sie sich im Bild 6-2 genauer ansehen.

Zuerst wird aus dem Operanden (hier wieder $58, 88 dez.)
und dem Offset eine Zero-Page Adresse ermittelt, im

Beispiel wäre das $58 + $04 = $5C. Falls sich dabei ein Ergebnis größer als $00FF einstellen würde, so kommt es zur Umfaltung (s.o.). Aus dem Inhalt der so ermittelten Adresse (und der darauf folgenden) wird der Zeiger zusammengestellt, über den schließlich indirekt zugegriffen wird.

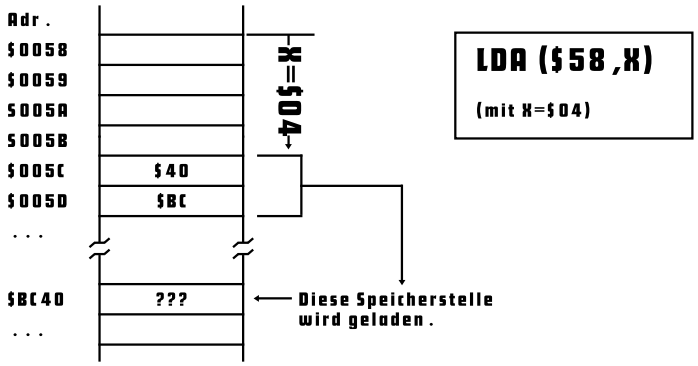


BILD 6-2: Indiziert-Indirekt

Sie sollten sich den Unterschied zwischen den beiden zuletzt besprochenen Adressierungsarten noch einmal deutlich vor Augen führen. Mit LDA ($C0),Y kann durch Verändern des Y-Registers ein Block von Speicherzellen beginnend ab der vom Zeiger in $C0, $C1 gegebenen Adresse angesprochen werden. Mit LDA ($C0,X) kann man dagegen durch Verändern des X-Registers eine Auswahl zwischen mehreren Zeigern (z. B $C0, $C1 bei X=0 oder $C2, $C3 bei X=2 etc.) in der Zero-Page vornehmen. Weitaus häufiger wird die erste dieser beiden Adressierungsarten verwendet, da sie sich ideal für Grafik-Manipulationen anbietet.
Als Operanden sind bei den letzten beiden Adressierungsarten nur Adressen in der Zero-Page möglich. Wohl können beliebige Adressen indirekt angesprochen werden, aber der Zeiger muß sich immer in der Zero-Page befinden. Befehle, die indirekt-indiziert arbeiten, sind daher zwei Bytes lang.

6.3 Spezielle Formen

Neben diesen allgemeinen Adressierungsarten gibt es noch drei besondere, die nur in Zusammenhang mit ganz bestimmten Befehlen verwendet werden dürfen.

Implied:
Befehle, die keine Adressangabe brauchen, bezeichnet man als Implied, was etwa bedeutet, daß die Adresse im Befehl schon eingeschlossen ist. Das ist zweifellos die einfachste Adressierungsart des 6502, und ein Beispiel haben Sie sogar schon kennengelernt: den BRK-Befehl. Implied Befehle sind 1-Byte Befehle, da sie nur aus dem Opcode bestehen.

Relativ:
Diese Adressierungsart läßt sich nur mit den bedingten Sprüngen anwenden und bewirkt einen Sprung relativ (daher der Name) zu momentanen Programmzähler.

Indirekt:
Es gibt sie also doch, die rein indirekte Adressierung. Aber leider nur für einen einzigen Befehl, nämlich dem unbedingten Sprungbefehl (JMP). Als Besonderheit ist der Operand dabei nicht an die Zero-Page gebunden, sondern darf eine beliebige Adresse sein. Daher ist diese spezielle Sprunganweisung auch ein 3-Byte Befehl.

Man kann mit Fug und Recht behaupten, daß der 6502 reichlich mit Adressierungsarten ausgestattet ist. Das macht die Sache für Anfänger nicht gerade einfach, aber Sie werden bald merken, daß es beim Programmieren von Vorteil ist, wenn man möglichst viele und mächtige Adressierungsarten zur Verfügung hat.

TEIL II: DER BEFEHLSATZ

In den folgenden Abschnitten werden alle Befehle des 6502-Assemblers mit den jeweils zur Verfügung stehenden Adressierungsarten vorgestellt. Soweit es sinnvoll ist, wird zu jeder Befehlsgruppe ein Programm-Beispiel gegeben, das den Befehl in der Praxis zeigt. Die Programme sind so kurz gehalten, daß es wenig Mühe bereiten sollte, sie unverzüglich einzutippen. Nutzen Sie diese Möglichkeit, Sie profitieren auf diese Weise viel mehr von diesem Buch, als wenn Sie es 'nur' lesen.

Bei jedem neuen Befehl werden Sie eine Tabelle mit all seinen Adressierungsarten finden. Daneben enthalten diese Tabellen aber noch mehr Informationen: In der Spalte SYNTAX ist die Schreibweise des Befehles in der jeweiligen Adressierungsart vermerkt. Dabei wurden folgende Abkürzungen verwendet:

Dat	:	Byte-Wert (Datum)	(0-255)
Adr	:	Adresse	(0-65535)
Zadr	:	Adresse der Zero-Page	(0-255)
Off	:	Offset bei Sprungbefehlen	(-128 bis +127)

Diese Abkürzungen stehen jedoch nur als Platzhalter. In einem Programm werden dort Konstante, symbolische Namen (Labels) oder mathematische Ausdrücke zu finden sein (s. Adressrechnung).

In der nächsten Spalte ('OPCODE') finden Sie den Code des Maschinenbefehles in hexadezimaler Schreibweise. Beim Programmieren mit ATMAS-II brauchen Sie diese Werte nicht zu wissen, denn ATMAS übersetzt die Befehlskürzel automatisch in diese Codes. Trotzdem sind Sie der Vollständigkeit halber mit angegeben. Die letzte Spalte sagt Ihnen schließlich, wie viele Bytes der Befehl benötigt. Diese Angabe ist zuweilen nützlich, besonders wenn es gilt, ein Programm für einen begrenzten Speicherbereich zu schreiben.

Im oberen Teil der Tabelle ist noch angegeben, welche Prozessor-Flags von dem Befehl verändert werden. Einige Befehle lassen die Flags auch unverändert, dort ist 'keine' vermerkt. Wenn Sie z.B.

lesen, dann bedeutet das, da der Befehl das 'Negativ-' und das 'Zero'-Flag neu stellt. Nach welchen Kriterien die Änderung der Flags erfolgt, ist jeweils beim Befehl angegeben.

I. Lade- und Speicherbefehle

Mit dieser Gruppe von Befehlen sind Sie bereits im letzten Kapitel in Berührung gekommen. In diesem Abschnitt sollen alle möglichen Adressierungsarten nebeneinander gestellt werden, damit Sie einen Überblick des Befehlsvorrates des 6502-Prozessors bekommen. Beginnen wir gleich mit den Ladebefehlen des Akkumulators:

Befehl: **LDA**		Flags: **N,Z**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Immediate	LDA #Dat	A9	2
Zero-Page	LDA Zadr	A5	2
Zero-Pg.,X	LDA Zadr,X	B5	2
Absolut	LDA Adr	AD	3
Absolut,X	LDA Adr,X	BD	3
Absolut,Y	LDA Adr,Y	B9	3
(Indir.),Y	LDA (Zadr),Y	B1	2
(Indir.,X)	LDA (Zadr,X)	A1	2

Anhand dieser Übersicht wird es schon klar, daß der Befehlssatz des 6502 nicht 'orthogonal' ist. Darunter versteht man, daß alle Befehle mit allen Adressierungsarten und allen Registern verwendet werden können, jedenfalls solange es eine sinnvolle Kombination gibt. Beim LDA-Befehl kann die indizierte Zero-Page Adressierung beispielsweise nur mit dem X-Register verwendet werden, aber nicht mit dem Y-Register. Es hilft also nur, wenn Sie sich die erlaubten Kombinationen einprägen, da es eine allgemeine Regel nicht gibt. Jeder Ladebefehl beeinflußt das 'Negativ'- und das 'Zero'-Flag, je nachdem, welcher Wert geladen wird. War es eine Null, so ist das 'Zero'-Flag gesetzt, könnte das Byte als negative Zahl betrachtet

werden, so ist das N-Flag nach dem Lade-Befehl gessetzt. Dieser Vorgang geschieht unabhängig davon, ob Sie vorhaben die Flags auch tatsächlich abzufragen. Auch den nächsten Befehl kennen Sie bereits aus dem einführenden ersten Kapitel. Er dient zum Ablegen des Akku-Inhaltes in einer Speicherzelle und trägt den Namen STA für 'Store Akkumulator'. Hier wieder die entsprechende Tabelle der Adressierungsarten:

Befehl: **STA**		Flags: **Keine**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Zero-Page	STA Zadr	85	2
Zero-Pg.,X	STA Zadr,X	95	2
Absolut	STA Adr	8D	3
Absolut,X	STA Adr,X	9D	3
Absolut,Y	STA Adr,Y	99	3
(Indir.),Y	STA (Zadr),Y	81	2
(Indir.,X)	STA (Zadr,X)	91	2

Bis auf 'Immediate' (ist hier nicht sinnvoll!) finden Sie bei STA die gleichen Adressierungsarten wie bei LDA. Die Flags werden jedoch nicht verändert. Lade- und Speicherbefehle gibt es in ähnlicher Form auch für die Index-Register X und Y. Dazu die nächsten Tabellen, wir beginnen mit dem X-Register:

Befehl: **LDX**		Flags: **N,Z**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Immidiate	LDX #Dat	A2	2
Zero-Page	LDX Zadr	A6	2
Zero-Pg.,Y	LDX Zadr,Y	B6	2
Absolut	LDX Adr	AE	3
Absolut,Y	LDX Adr,Y	BE	3

Mit LDX kann das Index-Register X geladen werden. Der Mechanismus gleicht dem des LDA-Befehles, wie dort werden

auch N- und Z-Flag berichtigt. Beachten Sie daher, daß die Flags nicht auf den Akku ausgerichtet sind, sondern von beliebigen Befehlen beeinflußt werden. Nach einem LDX-Befehl bleibt der Zustand der Flags solange bestehen, bis ein Befehl bearbeitet wird, der diese Flags erneut ändert. Als Offset für Indizierung kann (sinnvollerweise) nur das Y-Register verwendet werden, indirekte Adressierungsarten sind in diesem Fall überhaupt nicht zugelassen. Sie sehen schon, daß der Akku eine Sonderstellung besitzt, da das Gros der Adressierungsarten nur bei Befehlen erlaubt sind, die den Akku betreffen. Zum Speichern des X-Registers dient wie nicht anders zu erwarten der STX-Befehl:

Befehl: *STX* **Flags: Keine**

Adr.-Art	Syntax:	Opcode	Bytes
Zero-Page	STX Zadr	86	2
Zero-Pg.,Y	STX Zadr,Y	96	2
Absolut	STX Adr	8E	3

Beim STX-Befehl sind sogar nur drei Adressierungsarten möglich. Wie beim entsprechenden Befehl für den Akku werden die Flags bei STX nicht verändert. Nun bleiben noch die funktionsgleichen Befehle für das Y-Register zu besprechen. Wir beginnen wieder mit dem Ladebefehl:

Befehl: *LDY* **Flags: N,Z**

Adr.-Art	Syntax:	Opcode	Bytes
Immidiate	LDY #Dat	A0	2
Zero-Page	LDY Zadr	A4	2
Zero-Pg.,X	LDY Zadr,X	B4	2
Absolut	LDY Adr	AC	3
Absolut,X	LDY Adr,X	BC	3

Als Möglichkeit zur Indizierung ist in diesem Fall nur das X-Register zugelassen.

Die Befehle zum Speichern des Y-Registers:

Befehl: **STY**		Flags: **Keine**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Zero-Page	**STY Zadr**	**84**	**2**
Zero-Pg.,X	**STY Zadr,X**	**94**	**2**
Absolut	**STY Adr**	**8C**	**3**

Ein recht einfaches Beispiel zu diesen Befehlen haben Sie sicher noch aus dem einführenden Kapitel im Gedächtnis. Obwohl mit den wenigen Befehlen noch nicht viel anzufangen ist, soll trotzdem ein weiteres Beispiel folgen, das Ihnen den Einsatz dieser Anweisungen praktisch zeigt:

```
*
* Drei Farbregister laden
*
COLOR1  EQU $2C5
COLOR2  EQU $2C6
COLOR4  EQU $2C8
*
        ORG $A800
*
        LDA #$0A    hellgrau
        LDX #$00    schwarz
        LDY #$AS    hellgruen
        STA COLOR4  Rahmen
        STX COLOR1  Schrift
        STY COLOR2  Hintergrund
        BRK
```

In diesem Programm werden schon einige Fähigkeiten des ATMAS-II Assemblers verwendet. Nach den drei Kommentarzeilen werden per EQU-Anweisung drei Labels definiert, die den Farbregistern des Atari-Computers symbolische Namen zuordnen. Mit der ORG-Direktive wird die Anfangsadresse des Programmes festgelegt. Wie auch bei zukünftigen Beispielen werden Sie dort die Adresse $A800 finden, ab der der für Objektcode reservierte Bereich von ATMAS-II beginnt.
Es folgen die Assemblerbefehle, wieder wird zum Laden der Register die Immediate-, zum Speichern die absolute Adressierung verwendet. Durch den Einsatz der Labels wird

das Programm lesbarer, als wenn wir direkt die Adressen der Farbregister in den STA-Befehlen eingetragen hätten.

Der BRK-Befehl beendet das kleine Beispiel, und der Monitor von ATMAS-II zeigt Ihnen die Prozessor-Register an. Vergleichen Sie, ob im Akku, X- und Y- Register die im Programm festgelegten Werte zu finden sind! Zum Ausprobieren dieses Demos können Sie genau wie im Beispiel des Abschnittes 5 (Teil I) vorgehen.

2. Transfer-Befehle

Im Unterschied zu den Lade- und Speicherbefehlen dienen die Transferbefehle zum Austausch von Registerinhalten. Der Name 'Transfer' ist etwas unglücklich gewählt, denn tatsächlich werden die Registerinhalte kopiert. Man braucht Befehle aus dieser Gruppe häufig, wenn es um die Berechnung von Indices geht. Stellen Sie sich vor, es soll das Element 1+5 aus einer Tabelle geladen werden. Die Berechnung des Index 1+5 müssen Sie im Akku durchführen. Um aber die indizierte Adressierungsart einsetzen zu können, muß der Index vom Akku z.B. ins X-Register gebracht werden. Das wäre ein typische Aufgabe für den Befehl 'TAX' (zu lesen als 'Transfer Accu to X-Register').

Wie unschwer zu erkennen ist, geben der zweite und dritte Buchstabe die Quelle und das Ziel der Datenübergabe an. 'TYA' würde demnach den Inhalt des Y-Registers in den Akku kopieren. Anschließend eine Tabelle aller gültigen Kombinationen, da es alle diese Befehle jeweils nur in einer einzigen Adressierungsart gibt, wurden hier gleich mehrere Befehle in einer Tabelle zusammengefaßt:

Befehl: ***TAX,TXA,TAY TYA,TSX,TXS*** **Flags:** **N,Z***

Adr.-Art	Syntax:	Opcode	Bytes
Implied	TAX	AA	1
Implied	TXA	8A	1
Implied	TAY	A8	1
Implied	TYA	98	1
Implied	TSX	BA	1
Implied	TXS	9A	1

* TXS verändert KEINE Flags

Da die Transfer-Befehle das Ziel bzw. die Quelle der Daten im Befehlscode enthalten, braucht man keine weitere Speicheradresse angeben. Die Adressierungsart ist daher auch 'Implied'.

Die Abkürzung 'S' steht in diesem Zusammenhang für den Stackpointer, ein Register das für die Verwaltung des Stapels in Page 1 zuständig ist. Der 'TSX'-Befehl bietet

beim 6502 die einzige Möglichkeit den Wert Stackpointer zu lesen. Auf der anderen Seite kann der Stackpointer auch nur über das X-Register mit 'TXS' neu beschrieben werden. Im Regelfall ist es jedoch unnötig, den Stackpointer neu zu laden, da er vom Betriebssystem nach dem Einschalten des Computers initialisiert wird. Natürlich gibt es Fälle, in denen eine Veränderung des Stapelzeigers Vorteile bringt, aber solange Sie nicht mit allen Einzelheiten des Stapels vertraut sind, dürfen Sie sich nicht wundern, wenn der Computer bei Experimenten mit 'TXS' abstürzt. Der Stapel ist ein sehr empfindlicher Bereich des Rechners, da dort Adressen aufbewahrt werden, die zur Fortsetzung des Programmes nach der Bearbeitung von Unterprogrammen dienen.

Die Transfer-Befehle beeinflußen (ganz ähnlich zu den Lade-Anweisungen) das 'Negativ'- und das 'Zero'-Flag. Einzige Ausnahme ist der 'TXS'-Befehl, der keine Flags verändert.

Beispiel:

Hier wieder ein kleines Programmbeispiel: Alle Register (A, X, Y) sollen mit dem Wert $FF geladen werden. Das geht am schnellsten mit Transferbefehlen:

```
*
* Demo zu den Transfer-Befehlen
*
        ORG $AB00
*
        LDA #$FF
        TAX
        TAY
        BRK
```

Vergewissern Sie sich mit dem ATMAS-Monitor, ob der gewünschte Wert auch in den Registern enthalten ist!

3. Auf- und Abwärtszählen

Bestimmt haben Sie in BASIC-Programmen schon Ausdrücke wie X=X+1 verwendet. In Assembler gibt es Befehle, die ein Index-Register oder eine Speicherzelle in ganz ähnlicher Weise nach oben oder nach unten zählen lassen.

Sie können sich vorstellen, daß es dafür eine ganze Reihe von Anwendungen gibt: Schleifenzähler, Indices für Tabellen oder Felder und vieles mehr. Wieder gibt es pro Befehl nur die Adressierungsart 'Implied', so daß eine Tabelle genügt:

Befehl: ***INX,DEX INY,DEY*** **Flags:** **N,Z**

Adr.-Art	Syntax:	Opcode	Bytes
Implied	INX	E8	1
Implied	DEX	CA	1
Implied	INY	C8	1
Implied	DEY	88	1

INX (für Increment X-Register) bewirkt dabei die Erhöhung des X-Registers um eins, bei DEX (Decrement X-Register) wird das X-Register um den selben Betrag vermindert. Entsprechendes gilt für die Wirkung von INY und DEY auf das Y-Register. Da beide Register einen Zahlenbereich von $00 bis $FF (0 bis 255 dez.) darstellen können, tritt beim Überzählen die bekannte Umfaltung auf. Wenn das X- Register vor einem INX-Befehl den Wert $FF (255) enthielt, wird nach der Abarbeitung der Inhalt 0 zu finden sein. Genauso wird bei der Verminderung eines Index-Registers, das den Wert 0 enthält, wieder der Wert $FF erscheinen. Zwei Spezialfälle spiegeln sich in den Flags wieder: Ist der Inhalt eines Index-Registers nach einem obigen Befehl Null, dann ist das Zero-Flag gesetzt. Das Negativ-Flag wird dagegen gesetzt, wenn das Ergebnis negativ ist, d.h. das höchstwertige Bit des Resultats gesetzt ist. Andere Flags werden nicht beeinflußt.

Es wäre etwas wenig, wenn man nur die beiden 8-Bit Register X und Y als Zähler verwenden könnte. Man denke nur daran, wenn man mehrere Schleifen ineinander verschachteln will.

Mit zwei weiteren Befehlen kann man jede Speicherzelle herauf- oder herabzählen. Die Wirkung in Bezug auf die Umfaltung und die Prozessorflags ist völlig gleich mit den registerbezogenen Befehlen. Der Befehl zum Erhöhen von Speicherzellen heißt 'INC' (für Increment Memory).

Befehl: **INC**		Flags: **N,Z**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Zero-Page	INC Zadr	E6	2
Zero-Pg.,X	INC Zadr,X	F6	2
Absolut	INC Adr	EE	3
Absolut,X	INC Adr,X	FE	3

Entsprechend der DEC-Befehl (Decrement Memory) zum Abwärtszählen von Speicherzellen:

Befehl: **DEC**		Flags: **N,Z**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Zero-Page	DEC Zadr	C6	2
Zero-Pg.,X	DEC Zadr,X	D6	2
Absolut	DEC Adr	CE	3
Absolut,X	DEC Adr,X	DE	3

Beispiel:

Mit diesen zusätzlichen Befehlen kann man schon einen kurzen Text am Schirm erscheinen lassen. Im nachfolgenden Beispiel wird dies unter Umgehung des Betriebssystems bewerkstelligt. Die Codes der einzelnen Zeichen werden direkt in den Bildschirmspeicher geschrieben. Sie kennen diese Methode vielleicht von verschiedenen BASIC-Programmen. Man sieht schon, das Programm ist mit dem begrenzten Befehlsvorrat noch recht uneffektiv, besonders da noch keine Schleifen verwendet werden. Die Speicherstellen SAVMSC und SAVMSC+1 ($58, 59) bilden einen Zeiger des

Betriebssystems, der immer auf die erste Speicherzelle des Bildschirmspeichers deutet. Da dieser Zeiger sich in der Zero-Page befindet, können wir ihn gleich als Basisadresse für die indirekt-indizierte Adressierungsart verwenden.

```
*
*
* Einfache Textausgabe
*
SAVMSC  EQU $58 Zeiger auf Video-Ram
*
        ORG $A800
*
        LDY #0          Offset=0
        LDA #$21
        STA (SAVMSC),Y
        INY             Offset=1
        LDA #$34
        STA (SAVMSC),Y
        INY             Offset=2...
        LDA #$21
        STA (SAVMSC),Y
        INY
        LDA #$32
        STA (SAVMSC),Y
        INY
        LDA #$29
        STA (SAVMSC),Y
        BRK
```

Zuerst wird der Offset (in diesem Fall das Y-Register) auf Null gesetzt, damit der erste Speicherplatz des Video- Speichers, (die linke obere Ecke) adressiert wird. Der Akku wird mit dem Code $21 geladen, es handelt sich um ein 'A' in interner Darstellung, und dieses Zeichen wird per indirekt-indiziertem Store-Befehl im Speicher abgelegt. Nun wird der Offset mit INY auf eins erhöht, ein neuer Zeichencode geladen, und dieser in der nächsten Zeichen- position abgelegt und so weiter... Verfahren Sie mit dem Demoprogramm genauso wie im einleitenden Beispiel beschrieben.

4. SPRUNGBEFEHLE

Die wichtigsten Helfer zum Programmieren von Schleifen und Verzweigungen sind die in höheren Programmiersprachen so viel geschmähten Sprungbefehle. Da es in Assembler keine komfortablen IF . . . THEN- Verzweigungen, geschweige denn FOR . . . NEXT- oder gar WHILE-Schleifen gibt, muß man jede Verzweigung oder Schleife mit Sprungbefehlen programmieren. Das Ebenbild von GOTO heißt in Assembler 'JMP' (für jump, springe), an Stelle von IF. . . THEN tritt eine ganze Reihe von sogenannten BRANCH-Befehlen. 'Branch' steht hier für 'Verzweigung', also Befehle, die den Programmablauf in Abhängigkeit von bestimmten Bedingungen an anderer Stelle fortsetzen.

4.1 Unbedingte Sprungbefehle:

Jump Ein Sprungbefehl wird bei Maschinenprogrammen schon allein dadurch erreicht, daß eine neue 16 Bit Adresse in den Programmzähler PC geschrieben wird. Genau diese Aufgabe erledigt der Befehl JMP (für 'Jump'), dessen Tabelle der Adressierungsarten folgt:

Befehl: **JMP**		Flags: **Keine**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Absolut	**JMP Adr**	**4C**	**3**
(Indirekt)	**JMP (Adr)**	**6C**	**3**

Der JMP-Befehl kennt nur zwei Adressierungsarten: absolut und indirekt. Absolut heißt hier, daß das Programm an der im JMP-Befehl als Operand angegebenen Adresse fortgesetzt wird.

Beispiel

Wenn Sie das nachfolgende Programm wie gewohnt mit dem Monitor an der Adresse $A800 starten, werden Sie feststellen, daß der Computer den gewünschten Effekt zeigt, aber sonst keinerlei Tasten mehr annimmt. Und das ist ganz logisch, denn das Beispiel enthält eine Endlosschleife,

die der Prozessor aus eigener Kraft nicht mehr verlassen kann. Anders als in Basic hilft hier die BREAK-Taste nicht, in Maschinensprache muß man schon andere Geschütze auffahren. Nur ein Druck auf RESET erlöst den Computer, sie finden sich anschließend im Editor des ATMAS-Systems wieder.

```
*
* Spezialeffekt: Bildstoerung
*
RANDOM   EQU $D20A       Zufallszahl
COLPF2   EQU $D018       Hintergrundfarbe GR.0
*
         ORG $A800
*
ENDLOS   LDA RANDOM
         STA COLPF2
         JMP ENDLOS
```

Das Programm benutzt den in der Hardware eingebauten Zufallszahlengenerator, dessen Inhalt mit der Geschwindigkeit eines Maschinenprogrammes in das Farbregister geschrieben wird, das für den Hintergrund in GRAPHICS 0 zuständig ist. Aufgepaßt, hier wird nicht das Schattenregister in PAGE 2, sondern das Farbregister der Hardware hergenommen. Auf diese Weise kommen die unregelmäßigen Farbwechel innerhalb einer Bildzeile vor, die auch den Namen des Programmes geprägt haben.

Indirekter Jump

Als einziger Befehl erlaubt JMP die indirekte Adressierung, eine weitere Ausnahme im 6502-Befehlssatz ist, daß die Adresse nicht nur in der Zero-Page (wie bei den indiziert-indirekten Adressierungsarten) sondern im gesamten Adressbereich liegen darf. Mit dem indirekten JMP-Befehl ließe sich z.B. auch der ON . . . GOTO Befehl von Basic leicht ersetzen.

Beispiel: ON . . . GOTO

Im Programm-Beispiel (s.u.) wird je nach Anfangswert des X- Registers zu den Labels ZIEL1 (X=0), ZIEL2 (X=2) oder ZIEL3 (X=4) vezweigt. Probieren Sie es aus, indem Sie die Werte der Reihe nach in den LDX-Befehl einsetzen, assemblieren und das Programm im Monitor starten. Am Inhalt des Akkus (in der BRK-Meldung) können Sie erkennen, welcher Label angesprungen wurde.

Statt 1, 2, 3 darf man in diesem einfachen Beispiel nur die Werte 0, 2, 4 verwenden. Der Grund dafür ist in der Anordnung der Sprungziele in der TABELLE zu suchen, denn dort belegt die Adresse eines Labels je zwei Bytes. Im Programm wird das mit X adressierte Byte aus TABELLE in das LSB von VEKTOR geschrieben. Danach wird das nächste Byte aus TABELLE geladen und ins MSB von VEKTOR gebracht. Dies kann ohne Erhöhung des X-Registers geschehen, indem man die Basisadresse per Adressrechnung um eins erhöht (TABELLE+1) und dann mit X indiziert.

Die Tabelle der Sprungziele läßt sich leicht mit einer DFW-Direktive anlegen. ATMAS setzt später beim Assemblieren die tatsächlichen Adressen für die Labels ein. Nach dem Label VEKTOR legen wir mit DFW ein Wort (zwei Bytes) mit dem Anfangswert Null an, diese beiden Speicherzellen dienen als indirekter Zeiger für den Sprung. Beachten Sie, daß man den symbolischen Namen VEKTOR im Programm fast wie eine Variable verwenden kann. Vorsicht ist aber am Platze, wenn das MSB des Wortes angesprochen werden soll. Denn dazu muß man VEKTOR+1 schreiben.

```
*
* ON X GOTO ZIEL1,ZIEL2,ZIEL3
*
        ORG $A800
*
        LDX #2          Hier: 0,2 oder 4
        LDA TABELLE,X   Adresse in
        STA VEKTOR      Vektor ueber-
        LDA TABELLE+1,X tragen
        STA VEKTOR+1
        JMP (VEKTOR)
VEKTOR  DFW 0           Sprung-Vektor
*
* Tabelle der Sprungziele
*
TABELLE DFW ZIEL1,ZIEL2,ZIEL3

ZIEL1   LDA #1
        BRK
Ziel2   LDA #2
        BRK
Ziel3   LDA #3
        BRK
```

4.2 Bedingte Sprungbefehle: Branch

Ähnlich den IF-THEN Anweisungen in Basic gibt es in Assembler eine Reihe von 'bedingten' Sprungbefehlen. Wie nicht anders zu erwarten, arbeiten diese Befehle auf einem viel niedrigeren Niveau als die IF-Anweisung in BASIC. Sie erlauben nur die Abfrage einiger Flags des Status-Registers.

In den vorausgehenden Abschnitten wurde bereits mehrfach darauf hingewiesen, wie die einzelnen Befehle die Flags des Prozessors beeinflussen. Konnten Sie bisher mit diesen Informationen noch wenig anfangen, so werden Sie jetzt lernen, wie man mit Hilfe der Flags und den 'bedingten' Sprunganweisungen Verzweigungen und Schleifen programmieren kann.

Von den sieben vorhandenen Prozessor-Flags können vier gezielt abgefragt werden. Das sind:

Z: Zero-Flag.	Letztes Ergebnis war Null
N: Negativ-Flag.	Letztes Ergebnis war negativ (Bit 7=1)
C: Carry-Flag.	Überlauf aufgetreten.
0: Overflow-Flag.	Hinweis für Vorzeichen-Rechnung.

Die ersten beiden Flags sind Sie bereits bei den bisher besprochenen Befehlen begegnet. Das Carry-Flag wird für Rechen-Befehle wie Addition und Subtraktion benötigt und zeigt an, daß ein Überlauf über das achte Bit hinaus stattgefunden hat, anders ausgedrückt, wenn das Ergebnis größer als 255 ist. Wir werden uns mit diesem Flag bei den Befehlen zur Addition und Subtraktion noch näher auseinandersetzen.

Carry und Overflow

Das 'Overflow'-Flag wird dagegen nur benötigt, wenn man mit vorzeichenbehafteten Zahlen rechnen will. Sie erinnern sich: Das Vorzeichen steckt in so einem Fall im achten Bit. Eine Überlaufbedingung würde sich somit ergeben, wenn ein Übertrag vom siebten ins achte Bit erfolgt. Das Overflow-Flag wird in so einem Fall gesetzt und zeigt an, daß durch Überschreitung des Zahlenbereiches das Vorzeichen verändert wurde.

Im Anschluß finden Sie eine Tabelle, in der alle bedingten

Sprungbefehle, die speziell beim 6502 als 'Branch'-Befehle bezeichnet werden, zusammengefaßt sind:

Befehl: **BRANCH**		Flags: **Keine**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Relativ	**BEQ Off**	**F0**	**2**
Relativ	**BNE Off**	**D0**	**2**
Relativ	**BPL Off**	**10**	**2**
Relativ	**BMI Off**	**30**	**2**
Relativ	**BCC Off**	**90**	**2**
Relativ	**BCS Off**	**B0**	**2**
Relativ	**BVC Off**	**50**	**2**
Relativ	**BVS Off**	**70**	**2**

Aus der Tabelle kann man entnehmen, daß die bedingten Sprungbefehle nur zwei Bytes lang sind und weiterhin die 'relative' Adressierungsart benutzen. Das unterscheidet sie vom 'JMP'-Befehl, bei dem die absolute Sprungadresse angegeben wird. Die Branch-Befehle verlangen nur die Angabe des Abstandes zum Sprungziel, daher auch die Bezeichnung 'relativ'. Da der Abstand, in der Tabelle mit <Off> für Offset bezeichnet, nur ein einziges Byte groß ist, schränkt sich auch die Sprungweite auf einen 255 Bytes großen Bereich ein.

Relative Adressierung

Dazu kommt noch, daß man im Programm tunlichst nach vorne (zu höheren Adressen) und nach hinten springen möchte. Das wurde gelöst, indem der Offset als vorzeichenbehaftete Zahl angesehen wird, in der negative Zahlen im Zweierkomplement dargestellt sind. Da die Sprungweite vom Ende des Branch-Befehles gerechnet wird, sind Sprünge mit einer Weite von 129 Bytes in Vorwärtsrichtung und 126 Bytes zu niedrigeren Adressen möglich.

Es wäre recht nervenaufreibend, wenn man bei jedem Sprung diesen Offset per Hand ausrechnen müßte. Glücklicherweise geht Ihnen ATMAS-II hier hilfreich zur Hand. Sie tun bei einem Branch-Befehl einfach so, als wäre es ein 'JMP' und geben die absolute Sprungadresse an. ATMAS rechnet dies in die Sprungweite um, und meldet beim Assemblieren, wenn der maximal zulässige Bereich überschritten wurde.

Abfrage des Zero-Flags

Mit BEQ (Branch if equal) und BNE (Branch if not equal) wird das Zero-Flag abgefragt. BEQ testet dabei, ob das Z-Flag gesetzt ist, d.h. ob das Ergebnis des letzten Befehles Null war. Mit BNE wird genau das Gegenteil abgefragt, eben ob das Ergebnis ungleich Null war. Bei allen Branch-Befehlen gibt es diese zwei Variationen zum Abfragen beider Zustände. Anschließend finden Sie ein praktisches Beispiel zum Einsatz des BNE-Befehles. Aufgabe: Es soll ein Zeichen 256 mal in den Bildschirmspeicher geschrieben werden. Dazu wird ein beliebter Programmiertrick verwendet, den Sie in vielen Assemblerprogrammen wiederfinden werden: die Umfaltung. Erhöht man ein Index-Register über $FF (255 dez) hinaus, enthält es wieder die Null. Das wird im folgenden Beispiel benutzt:

```
*
* 256 Zeichen in Screen schreiben
*
SAVMSC  EQU $58         Zeiger auf Video-Ram
*
        ORG $A800
*
        LDY #0          Index-Reg. loeschen
        LDA #$21        B'-Code fuer 'A'
SCHLEIF STA (SAVMSC),Y  in Video-Ram
        INY             Index-Reg + 1
        BNE SCHLEIF     schon wieder Null?
        BRK             zum Monitor
```

Beim ersten Durchlauf wird die Null im Y-Register auf 1 erhöht, so daß der Sprung bei BNE durchgeführt wird. Erst wenn das Y-Register per Umfaltung wieder zu Null wird, ist das Kriterium des BNE-Befehls nicht mehr erfüllt und die Schleife wird verlassen.

Branch if Minus

BMI steht für 'Branch if minus', der Befehl prüft, ob das N- Flag gesetzt ist. Das Gegenstück dazu bildet die BPL- Anweisung (Branch if Plus), mit der abgefragt wird, ob das N-Flag rückgesetzt ist, d.h. das letzte Ergebnis positiv war.

Wie im vorherigen Abschnitt soll wieder ein (diesmal etwas längerer) Text am Bildschirm erscheinen. Wieder wird das Betriebssystem umgangen und die einzelnen Zeichen werden

direkt in den Bildschirmspeicher gebracht.

```
*
* Direkte Textausgabe mit Schleife
*
SAVMSC  EQU $58         Zeiger auf Video-Ram
*
        ORG $A800
*
        LDY #0          erstes Zeichen
SCHLEIF LDA TEXT,Y      Zeichen aus String
        BMI ENDE        Marke f. Ende?
        STA (SAVMSC),Y  in Video-Ram
        INY             naechstes Zeichen
        JMP SCHLEIF     schon alle?
*
ENDE    BRK
*
TEXT    ASC %Assembler ist nicht schwer%
        DFB $80
```

Sie sehen schon, durch die Einführung der Sprungbefehle werden die Maschinenprogramme eine Spur komplexer, aber auch kompakter. Wie im letzten Beispiel wird der SAVMSC-Zeiger auf das erste Zeichen des Bildschirmspeichers benutzt, unterschiedlich ist jedoch, daß diesmal ein ganzer String ausgegeben wird. Die Logik des Programmes erfordert es, daß dieser String mit einem Zeichen abgeschlossen ist, dessen höchstwertiges Bit gesetzt ist. Ein solches Zeichen, es wird im Beispiel durch DFB $80 erzeugt, wird vom Prozessor als 'negativ' erkannt und somit wird auch das entsprechende Flag gesetzt. Das wird wiederum benutzt, um das Ende des Strings zu bestimmen.

Die Logik des Programmes würde man in einer höheren Sprache als 'WHILE'-Schleife bezeichnen. Solange die Bedingung, d.h. solange das Zeichen 'positiv' ist, wird ein Zeichen per Index aus dem String gelesen und mittels indirekt-indizierter Adressierung in den Bildschirmspeicher geschrieben. Der Index (Y) wird um eins erhöht und die Schleife mit einem JMP geschlossen. Wenn die 'WHILE' Bedingung nicht mehr erfüllt ist, also das N-Flag nach dem Laden des Zeichens aus dem String gesetzt ist, dann wird zu dem Label ENDE verzweigt. Von dort geht es zum Monitor zurück.

Abfrage von OVERFLOW und CARRY

Bleiben jetzt noch die Branch-Befehle zum Carry- und Overflow-Flag. Analog lassen sich mit BCC (Branch if carry clear) das C-Flag auf Null und mit BCS (Branch if Carry Set) auf Eins-Status abfragen. BVC bedeutet folgerichtig 'Branch if oVerflow clear' und testet das O-Flag auf Null, BVS (Branch if oVerflow Set) testet Overflow auf Eins. Die Bedeutung dieser beiden Flags kann mit dem bisher besprochenen Befehlssatz allerdings noch nicht erklärt werden. Gedulden Sie sich daher bitte noch bis zu den nächsten Kapiteln.

5. Manipulationen an Flags

Eine ganze Reihe von Befehlen dient nur zur gezielten Veränderung der Prozessor-Flags. Der Zweck dieser Befehle liegt darin, entweder verschiedene Zustände des Prozessors aus- und einzuschalten, oder Voraussetzungen für andere Anweisungen zu schaffen. Hier eine Zusammenstellung der Befehle:

Befehl: **Flags**		Flags: **S. Text**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Implied	CLC	18	1
Implied	SEC	38	1
Implied	CLD	D8	1
Implied	SED	F8	1
Implied	CLI	58	1
Implied	SEI	78	1
Implied	CLV	B8	1

Die Befehle zum Setzen und Rücksetzen des Carry-Flags braucht man hauptsächlich zur Vorbereitung von Addition und Subtraktion. Das CARRY-Flag wird durch den Befehl CLC (clear carry flag) auf null und durch SEC (Set carry flag) auf eins gesetzt.

Weiterhin gibt es einen Befehl mit dem Kürzel 'CLV', der das Overflow-Flags auf Null setzt. Allerdings gibt es keinen ausdrücklichen Befehl, der dieses Flag in den Eins-Zustand bringt.

Flags als Schalter

Mit den restlichen vier Befehlen können bestimmte Zustände im Rechner umgeschaltet werden. 'SED' (Set Decimal Flag) schaltet von der binären Rechenweise auf BCD-Rechnung um, der Prozessor 'tut dann so', als ob er dezimal rechnen könnte. Auf die normale binäre Rechenweise läßt sich mit 'CLD' (clear decimal flag) zurückschalten.

Die beiden Befehle SEI und CLI kontrollieren eine sehr komplizierte aber auch mächtige Einrichtung des Atari-

Computers: den Interrupt. In Ihrem Atari-Computer gibt es neben dem 6502-Prozessor noch einige weitere hochintegrierte Bausteine, die für spezielle Aufgaben wie Grafik, Sound oder Ein-/Ausgabe zuständig sind. Diese Bausteine können den 6502 auffordern, sein momentanes Programm zu unterbrechen und kurzzeitig eine andere Routine zu bearbeiten. Es gibt einige Fälle, in denen ein solche Unterbrechung nicht erwünscht ist bzw. sogar zum Absturz des Rechners führen kann. Dazu kann man dann mit SEI (Set Interrupt disable) eine Sperre für Interrupts bewirken. Mit CLI wird dieser Zustand wieder aufgehoben.

Allerdings gibt es auch eine besondere Sorte von Interrupts, die sich um diese Sperre nicht kümmert (der NMI). Da die gesamte Materie der Interrupts sehr kompliziert und genauso wichtig ist, wurden im Anwendungs-Teil gleich zwei Abschnitte dafür reserviert. Bevor Sie mit SEI und CLI experimentieren, sollten Sie sich noch bis dort hin vorarbeiten.

Tips & Tricks

Zusammen mit den 'Branch'-Befehlen kann man die Befehle zur Veränderung der Flags auch zur Erzeugung von unbedingten Sprungbefehlen hernehmen. Beispiel:

```
...
CLC
BCC MARKEO
...
```

Sie fragen sich jetzt sicherlich, wozu das dienen soll - es gibt ja schon einen JMP-Befehl. Branch-Befehle haben aber den Vorzug, daß sie nicht auf absolute Adressen angewiesen sind. Ein Beispiel aus der Praxis: Ein kurzes Maschinenprogramm soll mit BASIC verwendet werden und in einem String untergebracht werden. Solche 'Kauderwelsch' Strings haben Sie bestimmt schon in der einen oder anderen Zeitschrift gesehen. Das Problem liegt nun darin, daß ein String nicht an einer festen Speicheradresse liegt, sondern je nach Länge des BASIC-Programmes an unterschiedlichen Adressen abgelegt wird. Ein JMP-Befehl würde aber immer an die gleiche Adresse springen und so bei Verschiebung des Strings zum sicheren Absturz des Computers führen. Der Trick mit den Branch-Befehlen springt dagegen über eine feste Distanz und ist an keine absolute Adresse gebunden. Programme, die auf diese Art geschrieben wurden nennt man 'relocatibel' (verschiebbar).

6. Addition und Subtraktion

Natürlich kann der 6502 auch rechnen. Allerdings muß man dabei mit geringem Komfort vorlieb nehmen, denn es gibt nur Befehle zum Addieren und Subtrahieren. Noch dazu ist der Zahlenbereich wieder auf ein Byte begrenzt. Will man größere Zahlen verarbeiten, so braucht man mehrere Programmschritte dazu. Ganz ähnlich ist es mit Multiplikation oder mit Division, für die es keinen eigenen 6502-Befehl gibt hier hilft nur, sich selbst ein kleines Programm zu schreiben.

6.1 Addition

Beginnen wir mit dem Befehl zum Addieren zweier Zahlen. Im 6502-Befehlssatz heißt er 'ADC', was für 'Add with Carry' also 'Addition mit übertrag' steht. Bei ADC wird immer das vom Operanden adressierte Byte zum Inhalt des Akkumulators addiert.

Befehl: *ADC* **Flags: N,Z,C,V**

Adr.-Art	Syntax:	Opcode	Bytes
Immediate	ADC #Dat	69	2
Zero-Page	ADC Zadr	65	2
Zero-Pg.,X	ADC Zadr,X	75	2
Absolut	ADC Adr	6D	3
Absolut,X	ADC Adr,X	7D	3
Absolut,Y	ADC Adr,Y	79	3
(Indir.),Y	ADC (Zadr),Y	71	2
(Indir.,X)	ADC (Zadr,X)	61	2

Insgesamt addiert dieser Befehl drei Komponenten: Den Akku-Inhalt, das vom Operanden adressierte Byte, sowie einen Übertrag im Carry-Flag, der von einer vorangegangen Rechnung stammen kann. Der Übertrag dient zum Addieren von Zahlen, die nicht mehr mit einem Byte allein dargestellt werden können.

Auf der anderen Seite hinterläßt ein ADC-Befehl auch seine

Spuren im Carry-Flag. Es wird gesetzt, wenn das Ergebnis der Addition größer als 255 ist. Mehr oder weniger kann man das C-Flag als das 'neunte Bit' des Akkumulators betrachten. Das Carry-Flag enthält sozusagen die Eins, die man beim Kopfrechen 'im Sinn' hat. Bei 14+9 würde man erst 4+9 berechnen, das gibt 3 und eine 'Eins im Sinn', die zur nächsten Stelle addiert wird. Genauso berechnet der Prozessor $F1+$E7 als $D8 und übertrag im Carry-Flag. Mit einem folgenden BCS können Sie somit feststellen, ob der Zahlenbereich bei einer Addition überschritten wurde.

Wichtig ist es auch, dieses Flag vor Beginn einer Addition mit dem CLC-Befehl auf Null zu setzen, denn sonst könnte ein von der letzten Addition zurückgebliebener übertrag das Ergebnis versehentlich verfälschen. Sie werden daher sehr oft auf derartige Befehlsfolgen stoßen:

```
CLC
LDA HILF
ADC #$40
STA HILF
```

Hier wird eine Konstante (im Beispiel $40) zu der Speicher zelle HILF, die schon zuvor definiert sein soll, addiert. Zuerst wird das Carry-Flag gelöscht, dann wird addiert. Typisch ist die Abfolge eines Lade-, eines Additions- und schließlich eines Speicherbefehls.

Mehrfach genaue Addition

In vielen Fällen reicht der Zahlenbereich von 0 bis 255 nicht aus. Man geht daher auf die Wort-Darstellung über, die insgesamt zwei Bytes belegt und sich bekanntlich zur Darstellung von Speicheradressen anbietet. Will man zwei solche Zahlen addieren, muß man sich einer mehrfachen Addition bedienen. Nehmen wir an, die beiden Worte WORT1 und WORT2 sollen addiert werden:

```
WORT1  DFW $2080
WORT2  DFW $3090

       ...
       CLC
       LDA WORT1
       ADC WORT2
       STA WORT1
       LDA WORT1+1
```

```
ADC WORT2+1
STA WORT1+1
...
```

Wie im Beispiel zuvor wird das Carry-Flag gelöscht, und dann die Addition der beiden niederwertigen Bytes ausgeführt. Sie erinnern sich doch noch, wie ein Wort beim 6502 abgelegt wird? Zuerst kommt das niederwertige Byte (LSB), danach folgt das höherwertige Byte (MSB). Aus diesem Grund wird mit WORT1 das LSB und mit WORT1+1 das MSB des Wortes angesprochen.

Nachdem die LSBs addiert wurden, beginnt das gleiche Spiel für die höherwertigen Bytes, nur mit dem Unterschied, daß kein CLC erfolgt. Klar, wenn bei der ersten Addition ein Übertrag auftrat, muß dieser bei der zweiter Addition berücksichtigt werden. Im obigen Beispiel wird zuerst $80 mit $90 addiert, das ergibt $110 oder $10 und 'Eins im Sinn' (C-Flag gleich eins). Bei der Addition der MSBs wird nun $20 plus $30 und der Übertrag der LSB-Addition berechnet, es ergibt sich $51. Als Wort dargestellt lautet Ergebnis somit $5110.

Dieses Schema ist natürlich nicht nur auf zwei Bytes beschränkt. Man kann durchaus auch Additionen über drei oder vier Bytes ausdehnen, und gewinnt auf diese Weise einen ziemlich großen Zahlenbereich. Bei vier Bytes wären das immerhin schon ein Bereich von 0 bis über vier Milliarden.

Overflow-Flag

Neben dem Carry-Flag werden vom Additionsbefehl auch Zero-, Negative- und das Overflow-Flag beeinflußt. N- bzw. Z-Flag werden nach schon beschriebenen Kriterien ermittelt, eine Besonderheit ist dagegen das Overflow- oder V-Flag. Mit diesem Flag wird eine Hilfe zum Rechnen mit vorzeichenbehafteten Zahlen gegeben. Sie wissen sicherlich noch, daß beim Zweierkomplement das höchstwertige Bit das Vorzeichen einer Zahl angibt. Bei einem Byte würde folglich das achte Bit das Vorzeichen darstellen. Tritt nun eine Überschreitung des zulässigen Zahlenbereiches auf, so würde bei der Addition des siebten Bits ein Überlauf ins Vorzeichen auftreten und dieses somit verändern.

Das Overflow-Flag würde in so einem Fall gesetzt werden, und zeigt Ihnen an, daß ein Überlauf des Zahlenbereiches aufgetreten ist, und somit das Vorzeichen nicht mehr stimmt. Sie können das Ereignis im Programm mit einem BVS-Befehl erkennen und geeignete Maßnahmen ergreifen (etwa

eine Meldung drucken, daß die Zahl nicht darstellbar ist). Bei einer mehrfach genauen Addition darf man das V-Flag nur bei der letzten Teiladdition beachten, denn das Vorzeichen steht im höchstwertigen und daher zuletzt addierten Byte.

Dezimale Addition

Durch das Setzen des D-Flags mit der SED-Anweisung wird der 6502 auf 'dezimale Rechenweise' umgeschaltet. Er addiert dann nicht mehr im Binärsystem, sondern benutzt Rechenregeln zu für BCD-Codes.

Je zwei dezimale Ziffern werden dabei in einem Byte dargestellt (s. Teil I, 2.6) der Zahlenbereich schränkt sich von 0-255 auf 0-99 ein. Ein überlauf im Carry-Flag stellt sich ein, wenn das Ergebnis einer Addition größer als 99 ist. Beispiel:

$34 + $21 = $55 Carry-Flag = 0
$57 + $85 = $42 Carry-Flag = 1

Das V-Flag hat in diesem Fall keine Bedeutung. Der 'dezimale Modus' wirkt nur auf die Befehle ADC und SBC (s. nächsten Abschnitt).

6.2 Subtraktion

Analog zum Additionsbefehl gibt es auch einen Befehl zur Subtraktion. Er heißt 'SBC', was für 'Subtract with Borrow' steht. Hier wieder die Tabelle der Adressierungsarten:

Befehl: **SBC**		Flags: **N,Z,C,V**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Immediate	SBC #Dat	E9	2
Zero-Page	SBC Zadr	E5	2
Zero-Pg.,X	SBC Zadr,X	F5	2
Absolut	SBC Adr	ED	3
Absolut,X	SBC Adr,X	FD	3
Absolut,Y	SBC Adr,Y	F9	3
(Indir.),Y	SBC (Zadr),Y	F1	2
(Indir.,X)	SBC (Zadr,X)	E1	2

Bei SBC-Befehl wird der Operand vom Akkuinhalt abgezogen, dabei wird ein eventuell von vorherigen Rechenoperationen vorhandener übertrag berücksichtigt. Der übertrag wird im Zusammenhang mit der Subtraktion als 'Borrow' (engl. für 'borgen') bezeichnet.

In der Schule lernt man ja auch, daß man bei Subtraktionen zuweilen eins von der nächsthöheren Stelle borgen muß. Beispiel: Bei 23 minus 9 müßten Sie erst 3-9 probieren, das geht aber nicht. Also wird eins von der Zehnerstelle 'geborgt', und man kann 13-9 zu 4 berechnen. Bei der Zehnerstelle bleibt noch eine Eins stehen, da wir dort schließlich geborgt hatten.

Borrow

Auch der 6502 muß eine Anleihe bei der höheren Stelle nehmen, wenn er ein größeres von einem kleineren Byte abziehen muß. Wie bei der Addition nimmt er dazu das Carry- Flag, nur wird es diesmal gerade verkehrt herum verwendet. Eine Null im Carry heißt, daß eins geborgt werden mußte. Daher muß vor einer Subtraktion zuerst das Carry-Flag mit dem SEC-Befehl gesetzt werden. Sonst nimmt der 6502 an, daß

in einer vorherigen Subtraktion ein Borrow aufgetreten ist, und zieht daraufhin eine Eins mehr ab.

Man merkt sich diesen etwas 'verdrehten' Sachverhalt am besten mit einer Eselsbrücke: Bevor etwas geborgt werden kann, muß es erstmal da sein. Daher mit SEC eine Eins in Carry schreiben. Nach der Subtraktion kann mit BCC abgefragt werden, ob ein 'Borrow' aufgetreten ist. Bildlich gesprochen wurde in diesem Fall mehr abgezogen als eigentlich da war. Typisch für eine Subtraktion ist folgende Sequenz:

```
SEC
LDA HILF
SBC #$40
STA HILF
```

Mehrfach genaue Subtraktion

Wie auch bei der Addition läßt der SBC-Befehl auch eine mehrfache Subtraktion zu. Der Ablauf entspricht ebenfalls dem Beispiel bei der Addition, nur muß hier SEC den Anfang bilden. Berechnet wird WORT1 minus WORT2, das Ergebnis wird in WORT1 abgelegt. Hier das Beispiel:

```
WORT1  DFW $5110
WORT2  DFW $3090

       ...
       SEC
       LDA WORT1
       SBC WORT2
       STA WORT1
       LDA WORT1+1
       SBC WORT2+1
       STA WORT1+1
       ...
```

Zuerst wird die Subtraktion der LSBs ausgeführt, dann folgen die MSBs. Zwischen den beiden Subtraktionen darf das Carry-Flag nicht verändert werden.

Beispiel

Nach so viel trockener Theorie brauchen wir wieder einmal ein handfestes

Beispiel. Gleich den Atari einschalten, ATMAS-II booten und das folgende Listing 'ZAHLENAUSGABE 1' eintippen. Sie werden gleich mit einem Problem konfrontiert, das in Assembler genau dann auftritt, wenn man eine Zahl am Bildschirm ausgeben will. Ein 'PRINT' im Sinne von BASIC kennt der 6502 schließlich nicht.

Da es sich um ein ganz einfaches Beispiel handelt, können hier nur Bytes ausgegeben werden, und die nur in die linke obere Ecke des Bildschirms. Im Beispiel ist die Zahl in einem DFB-Befehl am Ende des Programmes zu finden, Sie können diesen Wert selbstverständlich nach Belieben verändern (und dann neu assemblieren!).

Algorithmus

über das Thema 'Umrechnung von Zahlensystemen' (und um nichts anderes handelt es sich dabei) haben wir uns im Teil I schon Gedanken gemacht. Wir haben festgestellt, daß man zumindest dividieren können müßte - und dazu reicht der bisher besprochene Befehlsvorrat noch lange nicht. Es geht aber auch ohne, wenn man es nur etwas geschickt anstellt.

Man geht davon aus, daß man mit einem Byte keine größere Zahl als 255 darstellen kann. Wir subtrahieren nun von diesem Byte solange 100, bis ein 'Borrow' auftritt und merken uns die Anzahl der Subtraktionen in einem Zähler. Nun addieren wir wieder 100 (die zuviel abgezogen wurden) und haben damit eine Zahl, die nur noch Zehner und Einer-Stelle enthält. Auf der anderen Seite wissen wir durch den Zähler, welche Ziffer an die Hunderter-Stelle zu schreiben ist. Vom verbliebenen Rest wird nun solange 10 subtrahiert, bis wieder ein 'Borrow' auftritt, damit kennen wir auch die Ziffer an der Zehnerstelle. Den Rest könnte man nun gleich für die Einer-Stelle verwenden, aber da das Beispiel so eine schöne Schleife enthält, wird das Spiel einfach nochmal für die Eins wiederholt. Wenn es Ihnen gefällt, können Sie das kleine Programm so umschreiben, daß die Einerstelle direkt verwendet wird.

Tips & Tricks

Ein paar Tricks am Rande: Um eine Ziffer in den zugehörigen Code für den Bildschirmspeicher umzuwandeln braucht man nur $10 addieren. Das wird im Programm geschickt gelöst, indem der Zähler (im X-Register) für die Subtraktionen gleich bei dieser Zahl beginnt und somit den fertigen Code erzeugt. Im Beispiel wird eine Byte-Variable 'HILF' in Speicherzelle $FF der

Zero-Page definiert. SAVMSC ist wieder der Zeiger auf das erste Byte des Bildschirmspeichers. Das Y-Register hat gleich zwei Funktionen: es zeigt in die Tabelle TAB10 auf eine der Zahlen 100, 10 oder 1, die momentan subtrahiert werden soll, außerdem gibt Y den Abstand der Ziffer vom Bildschirmrand an.

Damit erkannt wird, wann die Hauptschleife des Programmes dreimal (für 100, 10 und 1) durchlaufen wurde, muß das Y- Register auf den Wert 3 geprüft werden. Das wird im Programm gelöst, indem Y in den Akku übertragen und dann die 3 abgezogen wird. Ist der Akku dann Null, so ist Y gleich drei. Im nächsten Abschnitt werden Sie einen für diesen Zweck günstigeren Befehl kennenlernen. Wenn Sie wollen, können Sie das Programm auf Diskette speichern, da ein ähnliches später nochmal vorkommt. Sie brauchen dann nur noch einige Zeilen zu ändern.

```
*
* DEMO zu ADC/SBC: Zahlenausgabe I
*
SAVMSC    EQU $58        Zeiger auf Video-RAM
HILF      EQU $FF        Variable in Zero-P.
*
          ORG $A800
*
START     LDA ZAHL       Zahl laden
          STA HILF       zwischenspeichern
          LDY #0         Stellenzaehler=0
NEUSTEL   LDX #$10       Code fuer '0'
          LDA HILF
NEUZIFF   SEC            SBC vorbereiten
          SBC TAB10,Y    Potenz abziehen
          BCC KLEINER    war zuviel! -->
          INX            Code erhoehen
          JMP NEUZIFF    nochmal probieren
KLEINER   CLC            zuviel abgezogene
          ADC TAB10,Y    Potenz addieren
          STA HILF       merken
          TXA            und die Ziffer
          STA (SAVMSC),Y anzeigen
          INY            naechste Stelle
          TYA            schon 3 Stellen?
          SEC            dazu 3 abziehen
          SBC #3         und auf Null pruefen
          BNE NEUSTEL    noch nicht Null-->
          BRK
*
TAB10 DFB 100,10,1
*
ZAHL      DFB 234        hier Zahl einsetzen
```

7. Vergleichsbefehle

Bisher konnten nur sehr einfache Bedingungen mit Hilfe der 'Branch'-Befehle abgefragt werden. Durch die 'Compare'- Befehle lassen sich zwei Zahlen miteinander vergleichen und feststellen, ob Gleichheit vorliegt, bzw. welche Zahl die größer oder die kleinere ist. Solche Vergleichsbefehle gibt es für den Akkumulator sowie für die beiden Index-Register.

Der Befehl CMP vergleicht das vom Operanden adressierte Byte mit dem Inhalt des Akkumulators. Zu Anfang die Tabelle mit den Adressierungsarten dieses Befehls:

Befehl: **CMP**		Flags: **N,Z,C**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Immediate	CMP #Dat	C9	2
Zero-Page	CMP Zadr	C5	2
Zero-Pg.,X	CMP Zadr,X	D5	2
Absolut	CMP Adr	CD	3
Absolut,X	CMP Adr,X	DD	3
Absolut,Y	CMP Adr,Y	D9	3
(Indir.),Y	CMP (Zadr),Y	D1	2
(Indir.,X)	CMP (Zadr,X)	C1	2

Der CMP-Befehl führt den Vergleich aus, indem er intern den Operanden vom Akkuinhalt abzieht. Im Unterschied zum SBC-Befehl wird das Ergebnis der Subtraktion jedoch nicht in den Akku gebracht, sondern nur die Flags entsprechend des Ergebnises gesetzt. Wäre zum Beispiel das Ergebnis Null, so ist das Z-Flag nach einem CMP gesetzt, obwohl der Akkumulator unverändert bleibt. In diesem Fall wären die beiden Zahlen gleich (Differenz ist Null). Mit BEQ nach dem CMP könnte man den Programmablauf verzweigen lassen und hat somit eine Möglichkeit, zwei Zahlen auf Gleichheit zu prüfen. Jetzt können Sie sich auch vorstellen, warum der Befehl zur Abfrage des Zero-Flags gerade 'BEQ' (verzweige bei Gleichheit) heißt. Ein Beispiel:

```
...
LDA HILF       Akku laden
CMP #64        ist gleich 64?
BEQ MARKE1     ja, verzweige –>
...
```

Durch die interne Subtraktion kann man auch feststellen, welche der beiden verglichenen Zahlen größer oder kleiner ist. Ist nämlich ein 'Borrow' aufgetreten, dann war die Zahl im Akku kleiner als die Vergleichszahl. In diesem Fall (siehe SBC!) ist das Carry-Flag null, daher wird die Kleiner-Bedingung mit BCC abgefragt. Im anderen Fall, d. h. wenn das C-Flag nach einem CMP eine Eins zeigt, ist der Akku-Inhalt größer oder gleich als die Vergleichzahl. Das können Sie sich leicht anhand des SBC-Befehles selbst überlegen. Sind beide Zahlen gleich, tritt kein Borrow auf, daher ist Carry auf eins (und das Z-Flag gesetzt). Auch das N-Flag kann in bestimmten Fällen zu Vergleichen herangezogen werden. Übrigens braucht vor einem CMP-Befehl das CarryFlag nicht wie bei SBC gesetzt zu werden, denn das erledigt CMP von selbst.

Vergleiche mit Index-Register

Vergleichsbefehle gibt es auch für die Index-Register, allerdings sind dabei weit weniger Adressierungsarten zugelassen. Diese Befehle sind trotzdem sehr nützlich, da man sich die Transfer-Anweisung in den Akku sparen kann, wie wir sie im Beispiel 'Zahlenausgabe' des letzten Abschnittes noch verwenden mußten. Sehen wir uns die Tabelle der Adressierungsarten für den CPX-Befehl (Compare Memory with Index X) an:

Befehl: ***CPX***		**Flags:** **N,Z,C**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Immediate	**CPX #Dat**	**E0**	**2**
Zero-Page	**CPX Zadr**	**E4**	**2**
Absolut	**CPX Adr**	**EC**	**3**

Funktionell gilt das gleiche wie für CMP, nur wird hier das adressierte Byte vom Inhalt des X-Registers abgezogen und die Flags dementsprechend gesetzt.

Einen gleichwertigen Befehl gibt es auch für das Y-Register.

Auch er beherrscht nur wenige Adressierungsarten:

Befehl: **CPY**		Flags: **N,Z,C**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Immediate	**CPY #Dat**	**C0**	**2**
Zero-Page	**CPY Zadr**	**C4**	**2**
Absolut	**CPY Adr**	**CC**	**3**

Beispiel

Mit den neuen Hilfsmitteln können wir das Programm zur Ausgabe von Zahlen schon wesentlich einfacher und übersichtlicher gestalten. Mit einem CMP-Befehl wird nun vorher geprüft, ob eine Subtraktion von 100, 10 oder 1 überhaupt möglich ist. Man umgeht damit die Addition des zuviel abgezogenen Wertes. Ebenso kann der Vergleich des Stellenzählers im Y-Register nun mit 'CPY' sehr viel eleganter und ohne Zerstörung des Akkus programmieren. Zur Abwechslung wird diesmal keine feste Zahl ausgedruckt, sondern ein Systemzähler (eine Art 'Uhr' im Atari-Computer) in einer endlosschleife ausgedruckt. Das Programm kann jederzeit mit RESET angehalten werden.

```
*
* DEMO zu CMP/CPY: Zahlenausgabe
*
SAVMSC    EQU $58        Zeiger auf Video-RAM
HILF      EQU $FF        Variable in Zero-P.
RTCLOK    EQU $12        Uhr im Atari
*
          ORG $A800
*
START     LDA RTCLOK+2   Uhr laden
          STA HILF       erst mal speichern
          LDY #0         Stellenzaehler=0
NEUSTEL   LDX #$10       Code fuer '0'
          LDA HILF
NEUZIFF   CMP TAB10,Y    Abziehen moeglich?
          BCC KLEINER    nein, zu klein -->
          SBC TAB10,Y    SEC braucht man nicht!
          INX            Code erhoehen
          JMP NEUZIFF    nochmal probieren
KLEINER   STA HILF       merken
          TXA            und die Ziffer
          STA (SAVMSC),Y anzeigen
          INY            naechete Stelle
          CPY #3         Y-Reg. gleich 3?
          BNE NEUSTEL    nein! -->
          JMP START      Endlosschleife

TAB10     DFB 100,10,1
```

8. Unterprogramme und Stack

Das Gegenstück zum GOSUB-Befehl in BASIC nennt sich im 6502-Assembler 'JSR', was für 'Jump to Subroutine' steht. In Verbindung mit der Anweisung 'RTS' (Return from Subroutine) lassen sich Programmteile schreiben, die nach ihrer Bearbeitung selbsttätig zum aufrufenden Befehl zurückkehren. Dazu wird im Unterschied zum JMP-Befehl beim JSR die sogenannte 'Rücksprungadresse' aufbewahrt, bei der am Ende des Unterprogrammes mit der Bearbeitung fortgefahren wird. Zur Aufbewahrung der Rücksprungadressen verfügt der 6502-Prozessor über eine Fähigkeit zur Bearbeitung eines 'Stapels', den wir uns zuerst genauer ansehen wollen.

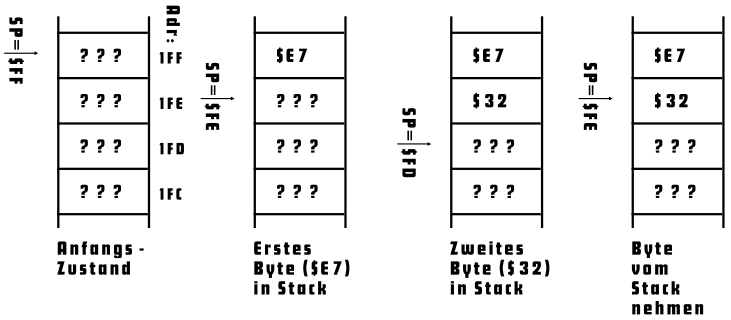
8.1 Der Stapel (Stack)

Beim Stack bietet sich ein recht anschaulicher Vergleich mit einem Kartenstapel (Talon) an. Dort ist es nur erlaubt, eine Karte oben aufzulegen bzw. die oberste Karte zu entnehmen. Legt man z.B. ein As, einen König und eine Dame in dieser Reihenfolge auf den Stapel, so taucht beim Abheben zuerst wieder die Dame, dann der König und schließlich wieder das As auf. In der Informatik bezeichnet man so einen Stapel, in dem das zuletzt hineingelegte Element wieder als erstes zum Vorschein kommt als 'LIFO-Stack' (für 'Last In - First Out'). Mit einem solchen Prinzip haben wir es auch beim Stapel des 6502-Prozessors zu tun. Hier kann jeweils ein Byte auf den Stapel 'geschoben' bzw. vom Stapel genommen werden.

Als Assemblerprogrammierer ist es sehr ratsam, über die Funktionsweise des Stapels sehr genau Bescheid zu wissen, da vielfach dessen innerer Aufbau trickreich zur Optimierung von Programmen eingesetzt wird. Zwei Komponenten sind zum Betreiben eines Stapels nötig: Ein reservierter Speicherbereich und ein Zeiger, der auf das nächste freie Element des Stapels zeigt. Für den Stack wird beim 6502-Prozessor die Speicherseite 1 (Page 1, von $0100 bis $01FF) benutzt. Diese Tatsache ist fest in der Hardware verankert und kann nicht vom Programmierer verändert werden. Die Verwaltung dieses Stack-Bereiches übernimmt der Ihnen schon bekannte Stack-Pointer (SP, Stapelzeiger).

Wie Akkumulator oder Index-Register ist SP ein acht Bit breites Register der 6502-CPU.

Im Gegensatz zum Kartenstapel wächst der Stack des 6502 nach 'unten', anders ausgedrückt, er wächst von hohen Adressen (ab $01FF) zu den niedrigen Adressen ($0100) hin. Daher wird der Stapelzeiger nach dem Einschalten des Computers durch eine Routine des Betriebssystemes ("Power-Up') auf den Wert $FF gesetzt und der Stack ist somit betriebsbereit. Die nachfolgenden Bilder zeigen Ihnen die Veränderung des Stack-Pointers sowie des Stapel-Speichers, wenn einzelne Bytes auf den Stack geschoben bzw. vom Stack genommen werden. Mit den Befehlen, die solche Operationen auslösen, werden wir uns in den nächsten Abschnitten befassen.



Interessant ist dabei, daß beim Herausnehmen eines Bytes nur der Stapelzeiger erhöht wird, das Byte selbst aber im Stack verbleibt. Es wird jedoch überschrieben, sobald ein neues Byte in den Stack geschoben wird.

8.2 Unterprogrammtechnik

Im folgenden wird vorausgesetzt, daß Sie wissen, wann und warum Unterprogramme benutzt werden. Hier gibt es auch keinerlei Unterschiede zu BASIC, wenn man davon absieht, daß die Befehle dort GOSUB und RETURN heißen. Auf der anderen Seite sollte man bei der Programmierung in Assembler einen Einblick in die internen Vorgänge des Prozessors haben und wissen, was bei der Ausführung eines JSR oder RTS geschieht. Denn oftmals werden diese Vorgänge

trickreich ausgenutzt, um die eine oder andere Schwäche des Befehlssatzes zu überbrücken. Doch zunächst die Tabelle der Adressierungsarten dieser beiden Befehle:

Befehl: **JSR,RTS**		Flags: **Keine**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Absolut	**JSR Adr**	**20**	**3**
Implied	**RTS**	**60**	**1**

Sie sehen schon, man braucht wahrhaft keine große Tabelle. JSR kann nur in einer einzigen, der absoluten Adressierung benutzt werden. RTS gibt es nur als 'Implied', das leuchtet ein, da die Adresse vom Stapel kommt.

Sehen wir uns jetzt die internen Vorgänge an, die bei einer Bearbeitung eines Unterprogrammes verrichtet werden. Dabei soll folgendes Programm zu Grunde liegen:

```
$A800          JSR UNTER
$A803          ...
...
$A900 UNTER    LDA #$01
...            ...
$A927          RTS
...
```

In diesem Beispiel ist am Anfang jeder Zeile die absolute Adresse mit angegeben, so daß Sie den Programmablauf verfolgen können. Gelangt der JSR-Befehl zur Ausführung, so werden zwei Vorgänge ausgelöst:

1.) Der momentane Programmzähler (PC) wird auf den Stack geschrieben. Da es sich beim PC um ein Wort (16 Bit) handelt, ist es nötig, zwei Bytes auf den Stack zu schieben. Dabei wird als erstes der höherwertige Teil von PC, dann erst des LSB berücksichtigt. Man muß ferner wissen, daß der PC zu diesem Zeitpunkt nicht mehr auf den Anfang des drei Byte langen JSR-Befehles zeigt, sondern bereits an dessen Ende (die Adresse $A802) angelangt ist.

2.) Erst jetzt wird die Adresse des Sprungzieles UNTER ($A900) in den Programmzähler geschrieben und somit der Sprung zum Unterprogramm ausgeführt.

Sobald das Unterprogramm seine Arbeit beendet hat, kehrt es mit einem RTS-Befehl zum Hauptprogramm zurück. Dazu wird die Rücksprungadresse wieder vom Stack geholt (zuerst LSB, dann MSB) und der Programmzähler geschrieben. Die Rücksprungadresse zeigte jedoch auf das letzte Byte des JSR-Befehles und NICHT auf den nach JSR folgenden Befehl. Aus diesem Grund erhöht RTS den Programmzähler noch um eins, bevor die Bearbeitung des Programmes (an der Adresse $A803) fortgesetzt wird. Der Stack hat nach dem RTS-Befehl den selben Zustand wie vor dem Aufruf des Unterprogrammes.

Verschachtelte Unterprogramme

Stellen Sie sich vor, innerhalb des Unterprogrammes UNTER würde erneut ein weiterer JSR-Befehl zur Unterroutine UNTER2 auftreten. Die neue Rücksprungadresse würde dann auf den Stack geschoben und vom Stapelzeiger unterhalb der vom ersten JSR angeordnet. Ist UNTER2 ordnungsgemäß mit einem RTS abgeschlossen, dann wird zuerst die Rücksprungadresse von UNTER1 vom Stack genommen, und später beim RTS von UNTER kommt wie beim Kartenstapel die ältere Adresse vom JSR UNTER zum Vorschein. Unterprogramme kann man so tief verschachteln, wie Platz für die Rücksprungadressen am Stapel vorhanden ist. Da beim 6502 insgesamt 256 Bytes für den Stack reserviert sind, darf die Verschachtelungstiefe bis zu 128 Ebenen betragen.

Um Mißverständnisse zu verhindern: Das heißt nicht, daß von einem Unterprogramm 128 andere aufgerufen werden dürfen. Vielmehr bedeutet dies, daß in einem Unterprogramm ein zweites, in diesem zweiten ein drittes . . . und im 126. Unterprogramm ein 127. -tes aufgerufen werden kann, bevor überhaupt das erste RTS erfolgte. Wenn Sie das jemals ausnutzen, dann Hut ab!

8.3 Direkte Benutzung des Stacks

Neben der Speicherung von Rücksprungadressen gibt es für den Stack auch noch eine andere Funktion: Er kann zur kurzzeitigen Ablage von Zwischenergebnissen benuzt werden. Zu diesem Zweck dienen die 'Push'- und 'Pull'-Befehle, mit welchen ein Byte auf den Stack gelegt bzw. vom Stack genommen wird.

Befehl: **PHA,PLA PHP,PLP**		Flags: **s . Text**	
Adr . -Art	**Syntax:**	**Opcode**	**Bytes**
Implied	**PHA**	**48**	**1**
Implied	**PLA**	**68**	**1**
Implied	**PHP**	**08**	**1**
Implied	**PLP**	**28**	**1**

Mit PHA (Push Accu) kann der Inhalt des Akkus am Stack abgelegt werden, Flags werden NICHT verändert. PLA (Pull Accu) bewirkt die Umkehrung und holt ein Byte vom Stack in den Akkumulator, wie beim LDA-Befehl werden hier jedoch N-und Z-Flag entsprechend des neuen Akku-Inhaltes gesetzt.

Durch den PHP (Push processor Status) wird das Status-Register (d. h. alle Prozessor-Flags) auf dem Stapel abgelegt, die Flags selbst bleiben aber unbeeinflußt. Wieder bewirkt PLP (Pull Processor status) das Gegenteil und holt ein Byte vom Stack in das Status-Register. Dadurch werden natürlich alle Flags verändert. Man kann diese beiden Befehle einsetzen, wenn man eine Rechenoperation oder einen Vergleich durchführt und die Auswertung erst an späterer Stelle im Programm vornehmen will. Im nachfolgenden Beispiel könnte man das zwar auch anders programmieren, aber es zeigt die Anwendung ganz gut:

```
...
CMP #15        Akku gleich 15?
PHP            Moment noch!
LDX#$FF        zuvor X=$FF
PLP            alte Flags
BEQ IST15      Akku ist 15–>
...
```

Nach der Besprechnung dieser Befehle wird Ihnen vielleicht auch klar, warum auf die genaue Beschreibung der Vorgänge bei JSR und RTS so großer Wert gelegt wurde. Mit einigen Push- und Pull-Befehlen kann man z.B. die von einem JSR stammenden Rücksprungadresse verändern oder gar entfernen. Das entspricht zwar nicht so ganz den Vorstellungen vom strukturierten Programmieren, erleichert aber so manches Problem. Gängige Programmierpraxis ist zum Beispiel die Simulation eines indirekten Sprunges durch ein RTS mit zwei PHA-Befehlen. Man legt dabei eine Adresse so auf den Stack, daß RTS eine Rücksprungadresse vermutet. Da RTS immer eine Eins zur Adresse addiert, muß man diese Eins zuvor abziehen. Da es sich hier meist um Adress-Tabellen handelt, legt man dort gleich deren Adressen minus 1 ab.

Beispiel

Als Beispiel verwenden wir wieder eine Simulation des BASIC-Befehles ON. . .GOTO, aber diesmal mit Trick 17! Da man bei dieser Methode keinen Zeiger im RAM braucht, bietet sich diese Methode für Programme an, die später in ein ROM (z.B. in ein Steckmodul) kommen sollen. Sogar im Betriebssystem Ihres Atari-Computers wird diese Technik verwendet!

```
**********************************
* ON X GOTO ZIEL1,ZIEL2,ZIEL3
*
* mit trickreicher Benutzung von RTS
**********************************
*
          ORG $A800
*
          LDX#4           Hier: 0,2 oder 4
          LDA TABELLE+1,X Adresse auf
          PHA             Stack schieben
          LDA TABELLE,X   zuerst MSB. dann
          PHA             LSB
          RTS
*
* Tabelle der Sprungziele (minus 1!)
*
TABELLE DFW ZIEL1-1,ZIEL2-1,ZIEL3-1

ZIEL1     LDA #1
          BRK
ZIEL2     LDA #2
          BRK
ZIEL3     LDA #3
          BRK
```

9. Logische Verknüpfungen

Neben Befehlen zum Rechnen besitzt die 6502-CPU auch drei Anweisungen zum logischen Verknüpfen von Zahlen. Diese Befehle gehen auf die 'Boolsche Algebra' zurück, falls Sie sich damit auskennen - um so besser. Für alle Leser, die mit Boolscher Logik noch nichts zu schaffen hatten folgt hier ein knapper Abriß.

9.1 Boolsche Algebra

Im Unterschied zur 'normalen' Algebra, die Verknüpfungen zum Rechnen bereitstellt, befaßt sich die Boolsche Algebra mit logischen Verknüpfungen. Die drei grundlegenden Verknüpfungen (ähnlich den Grundrechenarten) nennt man UND-, ODER bzw. NICHT-Verknüpfung. Alle anderen, auch noch so komplizierteren Logikbeziehungen lassen sich damit darstellen. Man kann auch alltägliche Bedingungen mit Hilfe dieser Verknüpfungen ausdrücken:

(Lampe brennt) = (Strom vorhanden)
UND (Schalter ein)
UND NICHT(Birne defekt)

In einer logischen Aussage ordnet man jedem Teil einen Wert zu, der nur zwei Zustände annehmen kann: wahr oder falsch. Im obigen Beispiel kann der Schalter nur entweder 'Ein' oder 'Aus' sein, eine andere Möglichkeit gibt es nicht. Diese beiden Zustände kann man genau in einem Bit darstellen, man bezeichnet einem Zustand mit '1' (z. B. Schalter ein), den gegenteiligen Zustand mit '0'.

Die einfachste Verknüpfung der Boolschen Algebra ist NICHT, durch sie wird einfach der Zustand umgedreht: aus 0 wird 1 und umgekehrt. Die beiden anderen Verknüpfungen verarbeiten je zwei Zustände zu einem neuen Wert. Man stellt solche Verknüpfungen gewöhnlich als 'Wahrheitstabellen' dar, wobei zu allen denkbaren Kombinationen der Eingangszustände der Ausgangswert angegeben wird:

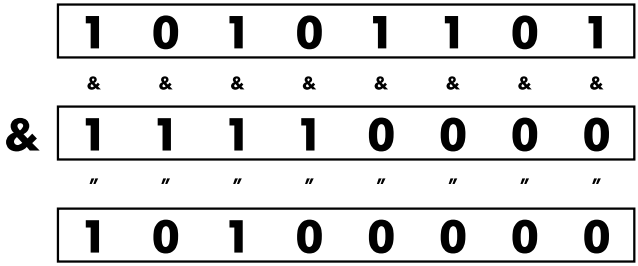
UND:			ODER:			EXOR:		
A	B	C	A	B	C	A	B	C
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Jetzt ist auch deutlich zu erkennen, woher die Verknüpfungen Ihren Namen haben. Bei UND nimmt das Ergebnis (C) nur den Wert eins an, wenn A UND B gleich eins sind. Auf der anderen Seite wird C bei der ODER-Verknüpfung zu '1', wenn A ODER B eine Eins enthalten. Wichtig ist, daß der Fall A=1 und B=1 auch die ODER-Bedingung erfüllt.

Die dritte Wahrheitstabelle nennt man EXOR bzw. Exklusiv-Oder Verknüpfung. Es handelt sich hier nicht mehr um eine 'Grundverknüpfung' sondern bereits um eine zusammengesetzte. Das Ergebnis von EXOR ist nur eins, wenn entweder A oder B gleich eins sind. Daß gerade diese Verknüpfung herausgegriffen wurde hat einen handfesten Grund - die CPU kennt einen Befehl, der die EXOR-Verknüpfung verwendet. Dafür gibt es keinen ausdrücklichen NICHT-Befehl. Das ist nicht weiter tragisch, denn EXOR enthält NICHT mit einem ganz einfachen Trick. Wenn man nämlich A fest auf eins hält, und unter dieser Voraussetzung B und C betrachtet (die beiden unteren Zeilen der Wahrheitstabelle) dann hat man genau eine NICHT-Beziehung zwischen B und C.

9.2 UND-Verknüpfung

Die drei zuletzt besprochenen Verknüpfungen spiegeln sich in drei Befehlen des Mikroprozessors wieder. Nur wirken diese nicht nur auf ein einziges Bit, sondern verknüpfen jeweils zwei Bytes miteinander. Dabei wird die Verknüpfung auf jede Bitstelle der beiden Bytes parallel angewendet. Stellen Sie sich das einfach so vor, wie wenn Sie zwei Bytes in binärer Schreibweise untereinander anordnen, und je zwei Bits gemäß der Wahheitstabellen miteinander verknüpfen. Hier ein Beispiel zur UND-Verknüpfung:



Aus diesen Beispiel sieht man auch gut, wozu die Logikbefehle meist verwendet werden. Man kann z.B. mit der UND-Verknüpfung gezielt einzelne Bits eines Bytes zurücksetzen, oder auch ganze Bit-Gruppen 'ausblenden'.

Den Befehl zur UND-Verknüpfung bezeichnet man beim 6502 mit dem Kürzel 'AND'. Der adressierte Speicherinhalt wird mit dem Akkumulator einer UND-Verknüpfung unterworfen, das Ergebnis bleibt im Akku zurück. Je nach Ergebnis werden Z-und N-Flag gesetzt, C- und V-Flag bleiben unverändert.

Befehl: *AND* **Flags: N,Z**

Adr.-Art	Syntax:	Opcode	Bytes
Immediate	AND #Dat	29	2
Zero-Page	AND Zadr	25	2
Zero-Pg.,X	AND Zadr,X	35	2
Absolut	AND Adr	2D	3
Absolut,X	AND Adr,X	3D	3
Absolut,Y	AND Adr,Y	39	3
(Indir.),Y	AND (Zadr),Y	31	2
(Indir. ,X)	AND (Zadr,X)	21	2

Das vom Operanden adressierte Byte bezeichnet man bei 'AND' (und auch bei den anderen logischen Befehlen) als eine 'Maske'. Das ist nichts weiter als eine binäre Zahl, in der die Bits, die rückgesetzt werden sollen mit einer Null gekennzeichnet werden, nicht betroffene Bits dagegen eine Eins bekommen. Beispiel: Das höchstwertige Bit (Bit 7) eines Bytes soll auf Null gesetzt werden. Die Maske würde in so einem Fall

%01111111 gleich $7F

lauten. Nehmen wir weiter an, die fragliche Zahl befände sich im Akku, so kann man das Problem mit Immediate-Adressierung lösen:

```
AND #$7F
```

Derartige Operationen braucht man oft bei Hardware- Registern, in denen jedes Bit eine andere Funktion hat. Denken Sie dabei nur an die Abfrage des Joysticks. Eine andere Anwendung wäre das Eintragen bzw. Löschen von Punkten in hochauflösender Grafik, denn dort gibt ein einziges Bit an, ob ein Punkt am Bildschirm an oder aus ist.

9.3 Die ODER-Verknüpfung

Nach demselben Prinzip arbeitet auch der Befehl ORA (OR Memory with Accumulator) nur wird hier eben das vom Operanden adressierte Byte mit dem Inhalt des Akkus 'ODER' verknüpft.

Befehl: *ORA* **Flags: N,Z**

Adr.-Art	Syntax:	Opcode	Bytes
Immediate	ORA #Dat	09	2
Zero-Page	ORA Zadr	05	2
Zero-Pg.,X	ORA Zadr,X	15	2
Absolut	ORA Adr	0D	3
Absolut,X	ORA Adr,X	1D	3
Absolut,Y	ORA Adr,Y	19	3
(Indir.),Y	ORA (Zadr),Y	11	2
(Indir.,X)	ORA (Zadr,X)	01	2

Im Unterschied zu AND wird ORA dazu benutzt, um einzelne Bits innerhalb eines Bytes auf eins zu setzen. In der Maske müssen alle Bits, die gesetzt werden sollen mit einer Eins markiert werden, alle nicht betroffenen erhalten eine Null. Im Beispiel sollen die vier niederwertigen Bits des im Akkumulator stehenden Bytes gesetzt werden:

```
ORA #$0F
```

Die binäre Darstellung von $0F lautet %00001111, wieder können mehrere Bits in einem Byte gleichzeitig beeinflußt werden. ORA eignet sich hervorragend zum Setzen einzelner Punkte in einem hochauflösenden Bildschirm.

9.4 EOR - Exklusiv Oder

Die Dritte im Bunde ist die Exklusiv-Oder Verknüpfung, der 6502-Befehl dazu lautet EOR. Es wird wieder der adressierte Speicher mit dem Akkumulator verknüpft. Das Ergebnis von EOR befindet sich wieder im Akku, Z- und N-Flag werden entsprechend gesetzt.

Befehl: *EOR* **Flags: N,Z**

Adr.-Art	Syntax:	Opcode	Bytes
Immediate	EOR #Dat	49	2
Zero-Page	EOR Zadr	45	2
Zero-Pg.,X	EOR Zadr,X	55	2
Absolut	EOR Adr	4D	3
Absolut,X	EOR Adr,X	5D	3
Absolut,Y	EOR Adr,Y	59	3
(Indir.),Y	EOR (Zadr),Y	51	2
(Indir.,X)	EOR (Zadr,X)	41	2

Während AND zum Löschen und ORA zum Setzen von Bits benutzt werden, kann man mit EOR den Wert der Bits umkehren (invertieren). Wieder braucht man eine Maske, in der alle zu invertierenden Bits mit '1', alle nicht betroffenen mit '0' bezeichnet werden. Im Beispiel sollen alle Bits des Akkus invertiert werden:

```
EOR #$FF
```

Alle Bits der Maske sind eins, so daß der gesamte Akku-Inhalt Bit für Bit invertiert wird. Befand sich z.B. $C1 im Akku, so findet sich danach $3E wieder. überlegen Sie sich dieses Beispiel durch Darstellung der Hex-Zahlen in binärer Form! Ein Programmbeispiel, in dem die gerade besprochenen Befehle benutzt werden, finden Sie im nächsten Abschnitt.

10. Bitverschiebungen

Nachdem Sie im letzten Abschnitt gesehen haben, wie einzelne Bits gezielt gelöscht oder gesetzt werden können, geht es nun um Verschiebungen von Bit-Stellen innerhalb eines Bytes. Diese Befehlsgruppe läßt sich in zwei Teile gliedern: 1) reine Schiebebefehle und 2) Rotationsbefehle, bei denen die Bits 'im Kreise' fließen. Man braucht diese Art von Befehlen hauptsächlich zur Programmierung von Multiplikationen und Divisionen.

10.1 Schiebebefehle

Zu dieser Gruppe zählen Befehle, die den Inhalt des Akkumulators oder einer Speicherzelle um eine Bit-Stelle nach links oder rechts verschieben. Dabei wird auf der einen Seite eine Null eingeschoben, während das herausgeschobene Bit ins Carry-Flag kommt. Der Befehl zum Links-Schieben heißt 'ASL' (Arithmetik Shift Left), der zur Verschiebung nach rechts nennt sich 'LSR' für Logical Shift Right.

Hier ein Diagramm, das die Vorgänge bei einer Verschiebung eines Bytes nach links mit ASL veranschaulicht:

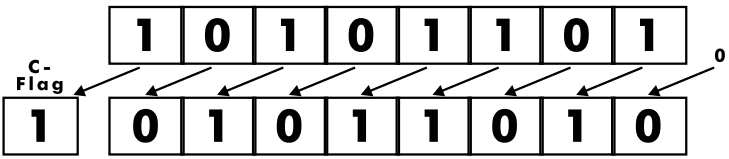


BILD 10-1: Arithmetic Shift Left (ASL)

Die Wirkung von ASL entspricht einer Multiplikation mit zwei. Warum? Denken Sie dazu nur ans Zehnersystem: Hängt man eine Null an eine Zahl hinten an, entspricht das einer Multiplikation mit 10. Da der 6502 im Zweiersystem arbeitet, resultiert aus dem Anhängen einer Null eben eine Multiplikation mit zwei. ASL kann mit impliziter Adressierung auf den Akku wirken bzw. eine Speicherzelle

bearbeiten.

Befehl: **ASL**		Flags: **N,Z,C**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Implied	**ASL**	**0A**	**1**
Zero-Page	**ASL Zadr**	**06**	**2**
Zero-Pg.,X	**ASL Zadr,X**	**16**	**2**
Absolut	**ASL Adr**	**0E**	**3**
Absolut,X	**ASL Adr,X**	**1E**	**3**

Das nach links herausgeschobene Bit wird ins Carry-Flag gebracht und kann dort abgefragt oder mit einem Rotationsbefehl in anderes Byte geschoben werden (s.u.). Negativ- und Zero-Flag werden gemäß des Ergebnisses berichtigt. Bei ASL und auch allen anderen Schiebe- und Rotationsbefehlen muß noch auf eine Besonderheit hingewiesen werden. Die Adressierungsart 'Implied' wird hier oft als 'Akkumulator' bezeichnet, da speziell der Akku adressiert wird. Daher muß bei einigen Assemblern (NICHT beim ATMAS-II) 'ASL A' statt 'ASL' geschrieben werden. Das 'A' steht in diesem Fall für Akkumulator. Diese Vereinbarung gilt auch für die anderen Befehle dieses Abschnittes.

LSR: Verschiebung nach rechts

Eine Verschiebung nach rechts kann mit dem Befehl LSR erreicht werden. Er arbeitet nach dem gleichen Muster wie ASL, nur wird diesmal die Null vom höchstwertigen Bit her eingeschoben, und das niederwertigste Bit landet im CarryFlag.

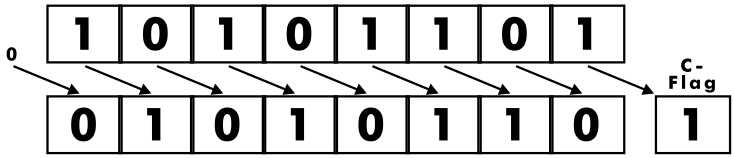


Bild 10-2: Logic Shift Right

LSR bewirkt somit eine Division durch zwei, das adressierte Byte wird 'halbiert'. das Carry-Flag enthält den Rest der

Divison.

Befehl: **LSR**		Flags: N,Z,C	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Implied	LSR	4A	1
Zero-Page	LSR Zadr	46	2
Zero-Pg.,X	LSR Zadr,X	56	2
Absolut	LSR Adr	4E	3
Absolut,X	LSR Adr,X	5E	3

Das N-Flag wird bei LSR immer auf Null gesetzt, da ja immer eine Null an die höchste Bit-Position eingeschoben wird. Ist das ganze Byte durch die Verschiebung zu null geworden, kann das mit dem Zero-Flag abgefragt werden.

10.2 Rotations-Befehle

Die letzten beiden Vertreter dieser Gruppe nennt man 'Rotations'-Befehle. Im Unterschied zu den reinen Schiebebefehlen wie wir sie bisher kennengelernt haben, wandern die Bits bei 'ROL' und 'ROR' im Kreise.

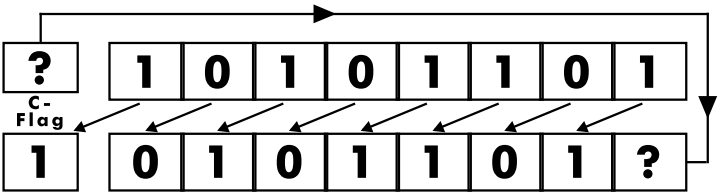


BILD 10-3 Rotate Left with Carry (ROL)

ROL entspricht weitestgehend der ASL-Anweisung, nur wird anstatt einer Null der Inhalt des Carry-Flags eingeschoben (im Bild mit '?' bezeichnet). Anschließend wird das achte Bit herausgeschoben und im Carry abgelegt. Bei der Ausführung eines weiteren ROL-Befehles, so würde dieses Bit an die erste Bit-Stelle gelangen. Das bedeutet, daß nicht acht sondern insgesamt neun Bits bei den Rotationsbefehlen beteiligt sind. Wieder wirken diese beiden Befehle entweder auf den Akkumulator oder auf ein beliebiges Byte

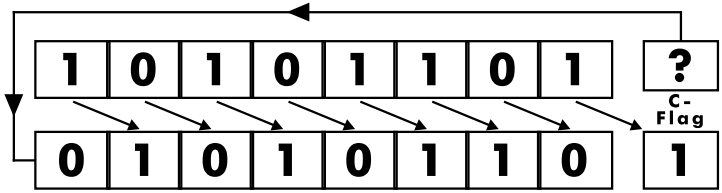
im Speicher.

Befehl: **ROL**		Flags: N,Z,C	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Implied	ROL	2A	1
Zero-Page	ROL Zadr	26	2
Zero-Pg.,X	ROL Zadr,X	36	2
Absolut	ROL Adr	2E	3
Absolut,X	ROL Adr,X	3E	3

Wie ASL entspricht ROL einer Multiplikation mit zwei, nur kann man mit Hilfe des C-Flags bestimmen, ob eine Eins oder eine Null in die freigewordene Bit-Stelle eingeschoben wird. Die Flags werden genau wie bei ASL beeinflußt.

Rotation nach rechts

ROR hingegen (ROtate Right with carry) wirkt ähnlich dem LSR-Befehl, nur wird der Inhalt des Carry-Flags an die bei der Verschiebung nach rechts freiwerdende Bit-Stelle eingeschoben.



Rotate Right with Carry (ROR)

Das herausgeschobene Bit wird im Carry-Flag abgelegt,ebenso kann es im N- und im Z-Flag zu Änderungen kommen.

Die Tabelle der Adressierungsarten zeigt nichts neues:

Befehl: **ROR**		Flags: N,Z,C	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Implied	ROR	6A	1
Zero-Page	ROR Zadr	66	2
Zero-Pg.,X	ROR Zadr,X	76	2
Absolut	ROR Adr	6E	3
Absolut,X	ROR Adr,X	7E	3

Sie werden sich jetzt vielleicht fragen, wozu denn so viele Befehle zum Verschieben und Rotieren nötig sind. Sie sind sehr nützlich, wenn man nicht nur ein einziges Byte sondern z. B. ein Wort um ein Bit verschieben will. Sehen Sie sich das gleich am nächsten Beispiel an: Ein Wort, bestehend aus zwei im Speicher nacheinander liegenden Bytes (LSB, MSB) soll mit zwei multipliziert werden.

```
ASL  WORT
ROL WORT+1
```

So funktioniert's: ASL verschiebt das LSB nach links, das niederwertigste Bit wird mit Null aufgefüllt. Nach ASL befindet sich das höchstwertige Bit des LSBs im C-Flag und wird nun per ROL an die unterste Stelle von WORT+1 (dem MSB des Wortes) geschoben. Das oberste Bit des MSBs wird im Carry-Flag hinterlegt. Wenn Sie dort eine Eins vorfinden, dann wissen Sie, daß das Ergebnis der Multiplikation den Zahlenbereich eines Wortes überschritten hat (überlauf).

Auf die gleiche Weise kann man auch durch zwei dividieren:

```
LSR  WORT+1
ROR WORT
```

Wissen Sie, warum man hier mit dem MSB beginnen muß?

Beispiel

Es ist soweit: Ihr Atari bekommt wieder Arbeit. Tippen Sie am besten gleich das nachfolgende Listing mit ATMAS-II ein. Es ist zwar nicht ganz so kurz wie die Beispiele zuvor, aber daran werden Sie sich gewöhnen müssen. Bei

Assemblerprogrammen liegt die Würze leider meist nicht in der Kürze. Das Programm zeigt jeweils eine Zeile eines GRAPHICS 0 Bildschirmes in inverser Darstellung an. Mit der SELECT-Taste können Sie die Markierung nach unten verschieben, nach der untersten Zeile wird wieder oben begonnen. Wenn es Ihnen Spaß macht, können Sie das Programm erweitern. Es eignet sich hervorragend z.B. zur Auswahl aus mehreren Menü-Punkten. Mit SELECT wird die Markierung (der inverse Balken) verschoben, die START-Taste könnte zum Auswählen des Menü-Punktes herangezogen werden. Sie kennen das Prinzip bestimmt aus verschiedenen kommerziellen Programmen (z.B. Print-Shop, tm Broderbund). Man müßte lediglich die Menü-Texte auf den Bildschim bringen, eine obere und untere Grenze für die Markierung festlegen und den Balken vielleicht nicht ganz so breit machen. Kein Problem für Sie, oder?

Werfen wir noch einen Blick in die inneren Vorgänge des Programmes. Der Anfang des Bildschirmspeichers wird wieder mit dem bekannten SAVMSC-Zeiger gefunden, die Funktionstasten werden mit dem Hardware-Register CONSOL abgefragt. Ist dessen Bit 1 (Maske %00000010) auf Null, so wurde die SELECT-Taste gedrückt.

Die Nummer der jeweils invertierten Zeile wird im X-Register festgehalten. In der Hauptscheife wird das zweite Bit von CONSOL mit AND isoliert und so kann abgefragt werden, ob SELECT betätigt wurde. Studieren Sie auch die Entprellung der Taste, sie ist wichtig, denn sonst würde der Balken über den Schirm flitzen, solange die Taste niedergehalten wird. Erst sobald Sie SELECT wieder loslassen, wird die bisher invertierte Zeile nochmal umgdreht (und damit wieder normal), die Zeilennummer im X-Register erhöht und die neue Zeile markiert.

Unterprogramme

Die harte Arbeit der Invertierung einer Zeile übernimmt das Unterprogramm INVERT. Es berechnet zuerst einen Zeiger auf den Anfang der durch X gegebenen Zeile. Eine Zeile am Bildschirm ist 40 Bytes lang, daher muß <X> mit 40 multipliziert werden (im Unterprogramm MUL40). Zum Ergebnis wird noch die Basisadresse des Video-Speichers aus SAVMSC addiert. Da der gewonnene Zeiger gleich in der Zero-Page abgelegt wurde, kann es von den indirekten Befehlen in der nachfolgenden Schleife benutzt werden. Die Invertierung geschieht mit einer Exclusiv-Oder Verknüpfung, die mit einer geeigneten Maske ($80) das höchstwertige Bit des Bild-

schirmcodes umkehrt.

Das Unterprogramm zur Multiplikation mit 40 verdient auch ein wenig Beachtung. Es handelt sich nicht um eine universelle Routine zur Multiplikation, vielmehr wurden nur Verdopplungen und Additionen eingesetzt. Sie können das anhand der Kommentare im Listing verfolgen. Natürlich eignet sich diese Methode nur zur Multipliktion mit (möglichst kleinen) Konstanten.

```
**********************************
* DEMO ZU LOGIK- UND SCHIEBE-BEFEHLE
*              V1.01
* Einzelne Zeilen eines GR.0-Screens
* werden invertiert. Bedienung mit
* der SELECT-Taste
* P.Finzel                      1986
**********************************
*
HILF      EQU $FE         Hilfs-Reg. in Zero-P.
*
* Hardware u. Betriebssystem
*
SAVMSC    EQU $58         Zeiger auf Video-RAM
CONSOL    EQU $D01F       Funktionstasten

          ORG $A800

          LDX #0          Start in Zeile 0
          JSR INVERT      invertieren

HAUPT     LDA CONSOL      SELECT-Taste ge-
          AND #2          drueckt?
          BNE HAUPT       nein --->
PRELL     LDA CONSOL      warten, bis Taste
          AND #2          wieder losge-
          BEQ PRELL       lassen --->
          JSR INVERT      alte Zeile normal
          INX             naechste Zeile
          CPX #24         war unterste?
          BNE WEITER      nein!
          LDX #0          sonst oberste Zeile
WEITER    JSR INVERT      neue Zeile invertieren
          JMP HAUPT       von vorne-->
```

```
*
* UNTERPGM: Eine Zeile invertieren
* Nummer der Zeile im <X>-Register
*
INVERT    JSR MUL40       Zeile mal 40
          CLC             und den Anfang
          LDA HILF        des Video-Speichers
          ADC SAVMSC      dazu addieren
          STA HILF
          LDA HILF+1      auch das MSB!
          ADC SAVMSC+1
          STA HILF+1
*
          LDY #39         nun die 40 Bytes
INV1      LDA (HILF),Y    der Zeile invertieren
          EOR #$80        hoechstes Bit
          STA (HILF),Y    umdrehen (Maske $80)
          DEY             weiter
          BPL INV1        schon alle?
          RTS             ja!

*
* Multiplikation mit 40
* Wert in <X>, Ergebnis in HILF, HILF+1
*
MUL40     TXA             Nummer der Zeile
          STA HILF        in Hilfs-Register
          LDA #0          MSB auf 0
          STA HILF+1
          ASL HILF        zuerst mit 4
          ROL HILF+1      multiplizieren
          ASL HILF        (* 2 * 2)
          ROL HILF+1
          CLC             jetzt Zeile noch-
          TXA             mal addieren ergibt
          ADC HILF        (Zeile*4)+Zeile=
          STA HILF        Zeile*5
          LDA HILF+1      das Carry nicht
          ADC #0          vergessen
          STA HILF+1
          ASL HILF        und noch mit 8
          ROL HILF+1      multiplizieren
          ASL HILF        ergibt Zeile*5*8
          ROL HILF+1      gleich Zeile*40
          ASL HILF
          ROL HILF+1
          RTS
```

Demo zu den Logik- und Schiebebefehlen

II. Sonstige Befehle

Außer den bisher besprochenen gibt es noch vier weitere Befehle, die sich in keine Schublade einordnen lassen. Dabei handelt es sich um einen Befehl, der rein gar nichts tut, zwei die sich mit Interrupts beschäftigen und einer recht vielseitigen Anweisung.

11.1 Der NOP-Befehl

Diese Anweisung erledigt - absolut gar nichts! Sollten Sie jetzt ins Grübeln kommen, wozu eine Anweisung gut sein soll, die einfach nichts tut, so lassen Sie sich folgendes sagen: NOP (steht übrigens für No OPeration) belegt immerhin ein Byte im Speicher und braucht zwei Arbeitstakte des Prozessors zur Ausführung. Somit kann man ein oder mehrere NOPs als Platzhalter für andere Befehle benutzen, die z.B. vom Programm selbst eingetragen werden könnten (selbstverändernder Code!).

Befehl:	NOP	Flags:	Keine
Adr.-Art	Syntax:	Opcode	Bytes
Implied	NOP	EA	1

Oft braucht man den NOP-Befehl auch beim 'Debugging', also bei der Fehlersuche in einem Programm. Wenn Sie sich ein Programm mit dem Monitor von ATMAS-II disassemblieren lassen und einen Fehler in einem ganz bestimmten Befehl vermuten, so kann es günstig sein, den Befehl zu Testzwecken einfach mal zu entfernen. Und das geht am schnellsten, wenn man ihn einfach mit dem Opcode des NOP-Befehles mit einem 'Change'-befehl des Monitors überschreibt, denn sonst müßte man ja das gesamte nachfolgende Programm verschieben (oder neu assemblieren). Sie sehen, es gibt eine ganze Menge Anwendungen für einen Befehl, der einfach nichts macht.

11.2 Software-Interrupt mit BRK

Diesen Befehl kennen Sie ja schon aus einigen Beispielprogrammen in diesem Buch. Nach Ihren bisherigen Erfahrungen bringt Sie BRK von einem Maschinenprogramm in den Monitor zurück und zeigt die Register des 6502-Prozessors am Bildschirm an. Diese Erklärung hat bisher ausgereicht, um Ihnen die Arbeit mit den Beispielen zu ermöglichen, doch nun ist es Zeit, die Vorgänge bei einem BRK etwas genauer zu beleuchten. BRK bezeichnet man als einen 'Software-Interrupt'. Rein vom Wort her betrachtet ist das so etwas wie ein 'schwarzer Schimmel', denn ein Interrupt (mehr darüber im Teil III) ist nun dadurch gekennzeichnet, daß er von der Hardware ausgelöst wird. Am nächsten kommt man der Funktion von BRK, wenn man sich einen Unterprogrammsprung mit fester Adresse vorstellt. BRK besorgt sich das Sprungziel aus zwei durch die Konstruktion des 6502-Prozessors fest vorgegebenen Speicherzellen. Diese liegen am Ende des Adressraumes an den Adressen $FFFE (LSB) und $FFFF (MSB). In Ihrem ATARI-Computer befindet sich an dieser Stelle das ROM des Betriebssystemes, in dem das Sprungziel für den BRK Befehl fest auf einen sogenannten 'Interrupt-Handler' eingeschrieben ist.

Wie auch bei einem Sprung zu einem Unterprogramm wird bei BRK eine Rücksprungadresse am Stapel hinterlassen, die es ermöglicht, nach Bearbeitung einer 'Interrupt-Routine' wieder so im Programm fortzufahren, als sei nichts geschehen. Merkwürdigerweise handelt es sich bei einem BRK jedoch um die Adresse des Befehles plus zwei. Darüber hinaus rettet der 6502 bei BRK den Inhalt des Status-Registers auf den Stapel und setzt zuvor noch das Break- Flag, um dem Interrupt-Programm zu signalisieren, daß der Interrupt nicht durch die Hardware, sondern per Programm ausgelöst wurde.

Befehl: ***BRK*** **Flags:** **B**

Adr.-Art	Syntax:	Opcode	Bytes
Implied	BRK	00	1

Der Interrupt-Handler im ROM des Ataris sieht in dem am Stack 'geretteten' Status-Register nach, und erkennt am B- Flag, daß der Interrupt von einem BRK kam. Er veranlaßt

daraufhin einen indirekten Sprung über einen Zeiger in Page 2 (VBREAK, $0206, $0207). ATMAS-II nützt dies aus, und schreibt in diesen Vektor die Startadresse eines Programmes, das den Inhalt der Prozessor-Register am Bildschirm ausgibt.

Daß als Opcode für BRK gerade die Null gewählt wurde, hat einen handfesten Grund. Wenn Sie sich mit EPROMs auskennen, dann wissen Sie, daß ein leeres EPROM nur 1-Bits enthält, die beim Programmieren auf Null gesetzt werden können. Umgekehrt kann ein 0-Bit nur durch löschen mit ultraviolettem Licht wieder auf eins gebracht werden. Sucht man in einem EPROM-Programm nach einem Fehler, so ist es technisch möglich, einen beliebigen Opcode mit einer Null überschreiben (alle Bits dieses Bytes auf Null setzten) und sich auf diese Weise mit einer geeigneten Interrupt-Routine die Register anzeigen zu lassen. Für Ihren ATARI-Computer hat das allerdings nur untergeordnete Bedeutung, da man selten mit EPROMs zu tun hat. Auf der anderen Seite läßt sich BRK beim ATMAS-II hervorragend zur Fehlersuche einsetzen.

Sie brauchen nur an einer suspekt erscheinenden Stelle einen Befehl mit einer Null (z. B. im Monitor) überschreiben. Sobald der Ablauf des Programmes an diese Stelle kommt, wird das Programm unterbrochen und Sie sehen die Register und Flags am Bildschirm. Im Computer-Jargon nennt man so etwas einen 'Break-Point'.

11.3 Rückkehr vom Interrupt mit RTI

RTI steht für 'Return from Interrupt' und hat eine ähnliche Wirkung wie der RTS-Befehl. RTI schließt eine Interrupt-Routine ab, holt das vom Interrupt gerettete Status-Register sowie die Rücksprungadresse vom Stack und setzt die Bearbeitung des unterbrochenen Programmes fort. Wenn Sie sich mit Interrupts noch nicht auskennen, dann haben Sie bitte bis zum Teil III Geduld: dort werden Ihnen die Interrupts des ATARI-Computers eingehend erläutert.

Befehl:	***RTI***	**Flags:**	**alle**
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Implied	**RTI**	**40**	**1**

Allerdings sollte man bei einer BRK-Routine mit RTI vorsichtig sein, da BRK die Adresse des Break-Befehles plus zwei am Stack hinterläßt. Hier muß der Stack erst 'per Hand' in Ordnung gebracht werden. Praktische Einsatzmöglichkeiten für RTI werden Sie im Teil III noch kennenlernen.

11.4 BIT - Der Alleskönner

BIT ist ein Befehl, mit dem Sie selbst so manchen Assembler-Freak noch zum Staunen bringen können. Er erledigt gleich mehrere Funktionen: Das Bit 7 des adressierten Bytes wird ins N-Flag, das Bit 6 ins V-Flag übertragen. Weiterhin wird das adressierte Byte mit dem Inhalt des Akkumulators UND-verknüpft (s. AND-Befehl), falls das Ergebnis Null ist, wird das Zero-Flag gesetzt. Das Ergebnis der UND-Verknüpfung wird jedoch verworfen, d. h. der Inhalt des Akkus wird NICHT verändert. Ganz schön viel auf einmal, oder?

Befehl: **BIT**		Flags: **N,Z,V**	
Adr.-Art	**Syntax:**	**Opcode**	**Bytes**
Zero-Page	**BIT Zadr**	**24**	**2**
Absolut	**BIT Adr**	**2C**	**3**

Damit liegt das Einsatzgebiet des BIT-Befehles bereits fest: Er dient zum Überprüfen von einzelnen Bits, wie auch der Name unschwer erraten läßt. Am schnellsten lassen sich die Bits 6 und 7 erfassen, da diese direkt in V- und N-Flag transferiert werden. Will man ein anderes Bit testen, muß erst eine passende Maske (in der nur das zu testende Bit auf 1 ist) in den Akku geladen werden. Dazu ein Beispiel, in dem die SELECT-Taste abgefragt wird.

```
LDA #$02      Maske f. Bit 2
BIT CONSOL    Bit testen
BEQ SELECT    Bit 2 = 0 –>
...
```

Das Zero-Flag wäre gesetzt, wenn das Ergebnis der UND-Verknüpfung des Inhaltes von CONSOL und $02 (binär:

%00000010) gleich Null wäre. Deswegen testet die Abfrage mit BEQ, ob Bit 2 rückgesetzt ist. Besonders günstig ist es, wenn die interessierende Information im Bit 6 oder 7 eines Bytes steht:

```
BIT ADRESSE    Bit 6 &7 testen
BMI MARKE1     Bit 7 = 1 -->
BVS MARKE2     Bit 6 = 1 -->
...
```

In diesem Fall muß nicht einmal der Inhalt des Akkus verändert werden! Leider liegen in den Hardware-Registern des ATARI-Computers wenig Informationen in diesen Bits, so daß der BIT-Befehl nicht oft zum Einsatz kommt. Man kann jedoch eigene Programme sehr elegant aufbauen, wenn man diese Eigenheit des 6502-Prozessors kennt.

TEIL III: ANWENDUNGEN

Der dritte Teil dieses Buches befaßt sich mit der Anwendung der Assemblerbefehle des 6502-Prozessors in der Praxis. Dazu werden erst einige Tips zur Form eines Assemblerprogrammes gegeben, dann auf spezielle Programmiertechniken hingewiesen, die in vielen 6502-Programmen eingesetzt werden. Anschließend finden Sie vier Anwendungen, die die Verwendung von Assemblerprogrammen mit speziellen Funktionen des Atari-Computers zeigen. Bei diesen Gelegenheiten werden kurze Einführungen in das Betriebssystem, die Player-Missile Graphik und in die Programmierung von Interrupts gegeben.

I. Programmiertechniken

1.1 Form eines Assemblerprogrammes

Assemblerprogramme neigen meist dazu, lang und unübersichtlich zu werden. Sie sollten dem gleich von vorn herein entgegentreten, und Ihre Programme so übersichtlich wie möglich gestalten. Auch wenn Ihnen das am Anfang etwas Überwindung kostet, es zahlt sich beim ersten größeren Projekt mit Sicherheit aus.

Programmkopf

Jedes Asemblerprogramm sollte mit einem Programmkopf beginnen, der alle wesentlichen Informationen zur Funktion enthält. So etwa den Titel, den Autor, das Datum der letzten Änderung und, sehr empfehlenswert, eine Versionsnummer. Diese Nummer setzt man am Anfang, d.h. wenn man das Programm gerade anfängt auf 1.00 und erhöht sie bei jeder Änderung. Die Betonung liegt dabei auf JEDER, sei die Änderung auch noch so geringfügig. Bei kleinen Änderungen erhöht man die letzte Stelle (1.01), bei größeren erhöht man die mittlere Stelle und setzt die letzte wieder auf Null (1.10), bei totalen überarbeitungen vergibt man eine neue Nummer vor dem Punkt (2.00). Das ist ein Konvention, die viele Programmierer verwenden. Der Sinn dieser Nummern: Oft hat man mehrere Versionen des gleichen Programmes gespeichert (z.B. als Sicherungskopie). Dann kann es schon

mal passieren, daß man eine alte Version erwischt und plötzlich tauchen Fehler auf, die schon längst beseitigt waren. Mit einer penibel durchgezogenen Versionsnummerierung passiert so etwas nicht.

Kommentare

Sparen Sie niemals mit Kommentaren. Wie Sie wissen, brauchen solche Anmerkungen keinen Speicherplatz im endgültigen Programm. Das heißt: Jeden wichtigen Befehl kommentieren, zwischen zwei funktionalen Blöcken ein paar erklärende Kommentarzeilen einschieben und schließlich jedem Unterprogramm einen 'Kopf' geben. Dort sollte der Name des Unterprogrammes, ein paar Worte zur Funktion und, ganz wichtig, die Ein- und Ausgaben notiert werden. Beipiel:

```
**************************
* UP MUL : 8-Bit Multiplikation
* Eingaben: <A>, <X>:Faktoren
* Ausgabe : <A> :Ergebnis
**************************
```

Wenn jedes Unterprogramm (UP) einen derartigen 'Kopf' besitzt, gewinnt ein Programm unglaublich an Lesbarkeit. Das zahlt sich auch aus, wenn man ein eigenes Programm nach Ablauf eines halben Jahres wieder aus der Versenkung holen muß. Dann nämlich hat man meist die Einzelheiten schon längst wieder vergessen, und hat Mühe, sein eigenes Programm zu entziffern. Ein gut dokumentieretes Listing hat man dagegen schnell wieder im Griff. Gleiches gilt verstärkt auch, wenn man Programme im Quellcode veröffentlichen möchte.

Verständliche Labels

Sie sollten immer versuchen, anstatt blanker Zahlen symbolische Namen zu verwenden. Es dreht einem immer wieder den Magen herum, wenn man Listings wie

```
LDA 53770
STA 709
JSR $E456
```

zu sehen bekommt. Hier helfen einige EQU-Anweisungen am Anfang des Programmes, und das Listing wird um Größenordungen verständlicher. Vergleichen Sie selbst:

```
RANDOM      EQU 53770
COLOR1      EQU 709
CIOV        EQU $E456
...
LDA RANDOM
STA COLOR1
JSR CIOV
```

Es gibt beim Atari sogar einen Satz festgelegter Namen für die Speicherzellen des Betriebssystemes (in Page 0, 2 und 3) sowie für die Hardware-Register. Sie finden diese in /2/ und /3/, außerdem wurden im Anhang einige Namen zusammengestellt, die in diesem Buch verwendet wurden. Wenn möglich, dann sollten Sie Sie nur diese Namen verwenden.

Es liegt natürlich in Ihrem persönlichen Ermessen, in wie weit Sie diese formalen Ratschläge beherzigen wollen. Bei kleineren Programmen kann man etwas großzügiger sein, bei größeren Projekten sollten Sie sich unbedingt daran halten. Und denken Sie immer daran, daß aus kleinen Projekten schnell größere werden können...

1.2 Verzweigungen und Abfragen

Aus dem Kapitel über 'bedingte Sprungbefehle' ist Ihnen sicherlich noch bekannt, daß Verzweigungen in Assembler eine ganz andere Technik als z. B. in BASIC erfordern. Anstatt komfortabler 'IF - THEN' Konstrunktionen müssen die Bedingungen in einzelne Schritte zerlegt und anhand von Flags abgefragt werden. In diesem Abschnitt werden wir uns mit Techniken zur Programmierung von Verzweigungen beschäftigen, die in der Praxis häufiger auftreten.

Besonders wichtig ist, daß Sie genau wissen, welche Flags von den einzelnen Befehlen verändert werden. So setzt z.B. der Lade-Befehl LDA bereits das Zero- und Negativ-Flag. Das bedeutet, daß eine Abfrage auf null (oder ungleich null) ohne Zwischenschaltung eines Compare-Befehles gleich nach dem Ladebefehl erfolgen sollte:

richtig:	überflüssig:
LDA RTCLOK	LDA RTCLOK
BEQ ISTNULL	CMP #0
	BEQ ISTNULL

Auf der anderen Seite heißt das natürlich auch, daß man aufpassen sollte, ob das gewünschte Flag auch durch den letzten Befehl beeinflußt wurde.

falsch:

```
INC ZEIGER
BCS UBLAUF
```

Die INC-Anweisung verändert nur Z- und N-Flag, nicht aber das Carry-Flag. An dieser Stelle kann gleich auf einen Fehler hingewiesen werden, der auch einem routinierten Assembler-Freak schon mal passieren kann:

```
falsch:          richtig:

INC ZEIGER       INC ZEIGER
CMP #100         LDA ZEIGER     (!)
BEQ GLEICH       CMP #100
                 BEQ GLEICH
```

Sehen Sie den Unterschied? INC (oder auch DEC) erhöht nur die Speicherzelle, der Akku bleibt unverändert. CMP dagegen wirkt nur auf den Akku, damit liefert das linke Beispiel ein falsches Ergebnis. Deshalb an dieser Stelle niemals das LDA vergessen.

Größer - kleiner

Größenvergleiche zweier Zahlen sind ein anderes Problem, das beim 6502 schweres Kopfzerbrechen verursachen kann. Bei der Besprechnung der Befehle ist Ihnen vielleicht aufgefallen, daß es keine ausdrücklichen Befehle wie etwa 'Springe wenn größer' gab. Zu diesem Zweck dienen die bedingten Sprungbefehle 'BCC' und 'BCS' in Verbindung mit Compare-Befehlen. Sehen Sie sich folgende Beispiele an:

```
Beispiel 1:      Beispiel 2:

LDA ZAHL         LDA ZAHL
CMP #100         CMP #100
BCC KLEINER      BCS GRGLCH
```

Der CMP-Befehl wirkt im Grunde wie eine Subtraktion, nur wird das Ergebnis verworfen und nur die Flags (Z,N und C) entsprechend dem Resultat gesetzt. Weiterhin ist zum Verständnis der Größenvergleiche wichtig, daß vor einem CMP oder auch CPX bzw. CPY das Carry-Flag immer automatisch gesetzt wird. Sie erinnern sich: bei der Subtraktion mußte dies mit SEC per Hand vorgenommen werden.

Im Beispiel 1 wird vom Akkuinhalt (der zuvor mit der Speicherzelle ZAHL geladen wurde) der Wert 100 abgezogen. War ZAHL kleiner als 100, so muß bei der Subtraktion 'eins geborgt' werden, d. h. das zuvor gesetzte C-Flag ist nun rückgesetzt. Aus diesem Grund kann hier mit BCC abgefragt werden, ob ZAHL kleiner als 100 ist.

MERKE: BCC verzweigt bei **KLEINER**

Nehmen wir nun an, ZAHL wäre größer als 100. Die Subtraktion ZAHL-100 benötigt keine geborgte Stelle, das bedeutet aber, daß Carry nach CMP noch gesetzt ist. Daher fragt BCS im Beispiel 2 die Bedingung 'größer gleich' ab. Gleich deswegen, da bei Zahl-100 ebenfalls kein 'Borrow' nötig wird und das C-Flag auch gesetzt wäre.

MERKE: BCS verzweigt bei **GRÖSSER-GLEICH**

Es lohnt sich, diese Sachverhalte genau zu durchdenken, weil man dabei ungeheuer leicht Fehler machen kann. Im Zweifelsfall kann man sich das mit Hilfe der Subtraktion und dem C-Flag überlegen.

Vergleiche bei 2-Byte Zahlen

Etwas problematisch sind auch Vergleiche zweier 16-Bit Zahlen. Hier heißt es genau aufgepaßt. Sehen Sie sich das folgende Beispiel an, in dem der Programmablauf verzweigt, wenn ZAHL1 kleiner als ZAHL2 ist:

```
ZAHL1     DFW 1100
ZAHL2     DFW 27000
...

          LDA ZAHL1+1  zuerst MSB
          CMP ZAHL2+1  pruefen
          BNE M1
          LDA ZAHL1    jetzt LSB
          CMP ZAHL2    vergleichen
M1        BCC KLEINER  -->
```

überlegen Sie sich einmal selbst, wie dieser Programmteil arbeitet. Mit ZAHL+1 wird das höherwertige Byte des 16-Bit Wertes angesprochen, aber das wissen Sie ja schon. Vielleicht noch ein Tip: Die niederwertigen Bytes muß man nur prüfen, wenn die MSBs gleich sind, oder?

1.3 Schleifen, Indexregister

Wichtigstes Hilfsmittel des 6502-Programmieres sind die beiden Indexregister. Mit ihnen kann man bequem Schleifen programmieren oder Tabellen und Felder verarbeiten. Auch hier gibt es wieder eine Reihe von speziellen Programmiertechniken, die im folgenden näher beleuchtet werden.

Nach Möglichkeit sollte man Schleifen so anlegen, daß der Index von höheren zu niedrigeren Werten gezählt wird. Man spart auf diese Weise den Vergleichsbefehl zum Erkennen des Schleifenendes, da man dazu direkt das Zero-Flag heranziehen kann. Hier zwei kleine Beispiele, die jeweils ein gedachtes Feld mit zehn Einträgen auf Null setzten:

abwärts zählen:

```
        LDA #0
        LDX #10
SCH1    STA FELD,X
        DEX
        BNE SCH1
```

aufwärts zählen:

```
        LDA #0
        LDX #1
SCH1    STA FELD,X
        INX
        CPX #11
        BNE SCH1
```

Noch etwas fällt beim Vergleich der beiden Schleifen auf: Damit in beiden Fällen der gleich Bereich (1 bis 10) abgedeckt wurde, mußten verschiedene Start bzw. Vergleichswerte herangezogen werden.

Bei der 'abwärts'-Schleife wurde bei 10 begonnen, d.h. der erste Eintrag in FELD erfolgt bei FELD+10. Der letzte Eintrag wird sinngemäß bei FELD+1 geschehen, denn danach wird X zu Null und der BNE-Befehl verzweigt nicht mehr. Die nach oben zählende Schleife beginnt ihre Arbeit bei FELD+1 und endet beim Eintrag in FELD+10, danach wird X auf 11 erhöht und die Schleife somit verlassen.

Element Null

In der Praxis tritt weit häufiger der Fall ein, daß eine Schleife von Null bis zu einem gewissen Wert durchlaufen werden soll. Ändern wir daher die beiden Beispiele so, daß jetzt die Feldelemente 0 bis 9 auf den Wert 255 gesetzt werden.

abwärts zählen:

```
      LDA #255
      LDX #9
SCH1  STA FELD,X
      DEX
      BPL SCH1
```

aufwärts zählen:

```
      LDA #255
      LDX #0
SCH1  STA FELD,X
      INX
      CPX #10
      BNE SCH1
```

Was hat sich nun verändert? Beim rechten Beispiel (aufwärts) wurden nur die Grenzen neu eingesetzt. Beim ersten Schleifendyrchlauf ist X gleich Null, beim letzten wird FELD+9 bearbeitet, der Wert bei CPX muß (wie oben) immer um eins größer gewählt werden.

Beim abwärts zählenden Beispiel wurde dagegen ein neuer Befehl zur Verzweigung gewählt. Der erste Durchlauf bearbeitet FELD+9, nun wird X immer weiter verringert, bis es schließlich nach einem DEX-Befehl zu Null geworden ist. BPL verzweigt aber trotzdem, da das N-Flag abgefragt wird - und das ist bei der Null eben noch gesetzt. Wenn im nachfolgenden Durchlauf die Speicherzelle FELD+0 bearbeitet wurde, überzählt schließlich der DEX-Befehl das X- Register, es wird zu $FF (255). Das höchstwertige Bit ist somit gesetzt, die Zahl kann als negativ interpretiert werden, und der BPL-Befehl findet seine Bedingung nicht mehr erfüllt. Damit wird die Schleife verlassen.

Vorsicht ist bei der letzteren Methode geboten, wenn der Index bei einem Wert größer 128 beginnen soll. Dann nämlich funktioniert es nicht mehr, da X von vorneherein als negativ (Bit 7 gleich 1) angesehen wird. Interessant ist jedoch, daß die abwärts zählende Schleife wieder die optimalere in Sachen Geschwindigkeit und Speicherplatz ist.

Verschachtelte Schleifen

Wie auch in BASIC können Schleifen ineinander geschachtelt werden. Dazu kann man z. B. die beiden Index-Register verwenden:

```
*
* Verschachtelte Schleifen
*
          ORG $A800
*
START     LDY #200         aeussere Schl. 200x
AUSSEN    LDX #0           innere Schl. 256x
INNEN     DEX              insgesamt 51200
          BNE INNEN        Durchlaeufe
          DEY
          BNE AUSSEN
          RTS zurueck      zu ATMAS-II
*
```

Dieses kleine Beispiel wäre eine kleine Zeitschleife, die wie eine leere FOR-NEXT-Schleife nichts tut, außer ein bißchen Zeit vergehen zu lassen. Es handelt sich um zwei ineinander geschachtelte Schleifen, die äußere benutzt das Y-Register, die innere das X-Register. Bei der inneren Schleife wird ein beliebter 6502-Trick verwendet: X beginnt bei Null, wird aber bereits im ersten Durchlauf überzählt (Umfaltung) und hat daher beim BNE-Befehl den Wert $FF. Nun ist es nicht mehr Null, BNE verzweigt solange, bis aus $FF wieder eine Null geworden ist. Dieser Kniff wird immer dann verwendet, wenn es gilt, genau 256 Durchläufe zu erzielen. Sie sehen: durch überzählen der Index-Register werden sehr elegante Konstruktionen ermöglicht.

Als Vertiefung zu diesem Abschnitt wurde noch eine universale Fill-Routine angefügt, die einen beliebigen Speicherbereich mit einem vorgegebenen Wert füllt. Hier wurde tief in die Trickkiste gegriffen, um das Programm in Punkto Geschwindigkeit zu optimieren. Versuchen Sie, die Funktion des Programmes im Listing zu verfolgen. Ein guter Assemblerprogrammierer muß sich auch in verzwickteren Programmen zurecht finden - hier haben Sie Trainingsmaterial. Am besten ist es, wenn Sie das Programm gleich mit ATMAS-II eintippen und ausprobieren.

```
************************************
*    FILL-ROUTINE MIT DEMO V1.1
*
*     Abbruch des Demos mit RESET
*
*   P. FINZEL               1986
************************************

SAVMSC    EQU $58          Zeiger auf Video-Ram
GROLEN    EQU 960          Anzahl Zeichen GR.0
ZEIGER    EQU $D4          Variable in Zero-Page
LAENGE    EQU $D6          -"-
WERT      EQU $D8          -"-
*
          ORG $A800
*
DEMO      LDA #0           Erstes Zeichen
          STA WERT
ENDLOS    LDA SAVMSC       Parameter
          STA ZEIGER       fuer FILL
          LDA SAVMSC+1     vorbereiten
          STA ZEIGER+1     Zeiger auf
          LDA #GROLEN:L    GR.0 Bildschirm
          STA LAENGE       960 Bytes
          LDA #GROLEN:H    beschreiben
          STA LAENGE+1
          JSR FILL         und aufrufen
          INC WERT         anderes Zeichen
          JMP ENDLOS       ==>

************************************
* FILL-ROUTINE: fuellt Speicherbe-
* reich mit festem Wert
*
* ZEIGER: Zeiger auf erste Zelle
* LAENGE: Anzahl der Bytes
* WERT  : einzuschreibender Wert
************************************

FILL      LDA WERT         Fuell-Wert
F3        LDY LAENGE+1     MSB der Laenge
          BNE FILMSB       noch nicht Null?
          LDY LAENGE       fertig?
          BEQ FERTIG       ja-->
          LDY #0
F1        STA (ZEIGER),Y   Rest gemaess
          INY              LSB fuellen
          CPY LAENGE
          BNE F1
FERTIG    RTS
*
* eine komplette PAGE fuellen
*
FILMSB    LDY #0           Trick mit
F2        STA (ZEIGER),Y   Ueberzaehlen
          DEY
          BNE F2
          DEC LAENGE+1     MSBs anpassen
          INC ZEIGER+1
          JMP F3           Schleife-->
```

2. Beispiele für Anwendungen

In den nächsten Abschnitten erhalten Sie die Gelegenheit, Ihre frisch erworbenen Kenntnisse der Assemblerprogrammierung am praxisnahen Beispiel zu vertiefen. Den Anfang macht ein Menue-Programm, eine Anwendung, die Sie sicherlich in eigenen Programmen oft einsetzen können. Danach folgt ein Beispiel zur Graphik-Programmierung, Stickwort Player-Missiles - ein heißes Thema für alle Spiele-Freaks.

Das sicherlich anspruchsvollste Beispiel behandelt die Interrupt-Fähigkeit des Atari-Computers, und zeigt Ihnen gleich zwei Anwendungen dafür. Da es besonders in Assembler wichtig ist, sich mit der 'Programmier-Umgebung', (darunter versteht man z.B. die Hardware-Register und Software im ROM) genau auszukennen, wird bei jedem Programm-Beispiel ein Themenkreis angesprochen. Im ersten Abschnitt werden wir einen Blick auf die Möglichkeiten der Ein-/Ausgabe mit dem Betriebssystem werfen, beim zweiten Beispiel die Zusammenhänge der PM-Graphik erklären und schließlich das Thema 'Interrupts' unter die Lupe nehmen. Natürlich kann in diesen Beispielen keiner der Themenkreise erschöpfend behandelt werden, das würde den Rahmen dieses Buches bei weitem überschreiten. Vielmehr wird ein Einblick in die Zusammenhänge gegeben, der Ihnen einen Einstieg ermöglichen soll.

2.1 Programmierung von Menüs

Leider nein - mit Gourmets und Restaurants hat diese Art von Menüs nur wenig zu tun. Bestenfalls mit der Speisekarte, denn auch dort müssen Sie unter vielen vorgeschlagenen Gerichten Ihre Auswahl treffen. Auch ein Computer-Menü hängt mit einer Auswahl zusammen, man braucht es immer dann, wenn der Benutzer eine Möglichkeit von mehreren wählen muß.

Die sicherlich einfachste Form eines Menüs besteht darin, daß man mehrere Wahlmöglichkeiten nummeriert am Bildschirm ausgibt, aus denen sich der Benutzer per Tastendruck eine aussuchen kann.

Ein Beispiel zu dieser Programmiertechnik kennen Sie bestimmt: Das 'DOS-Menü' zeigt Ihnen sämtliche vorhandenen Funktionen, von DOS 2.5, und Sie wählen eine davon mit den Tasten A bis P aus. Wir wollen natürlich nicht gleich ein ganzes DOS programmieren (das wäre dann doch ein bißchen zuviel), sondern das Prinzip an einem Beispiel zeigen, das die Farben des Bildschirmes per Menü-Auswahl ändert. Dabei werden Sie so nützliche Funktionen wie Ausgabe von Texten und Eingabe von Zeichen von der Tastatur kennenlernen.

Man kann sich vorstellen, daß die Programmierung eines Menüs in BASIC eine leichte Übung wäre. Einige PRINT- Befehle zur Ausgabe des Menü-Texte, ein GET-Befehl zum Lesen der Tastatur... fertig! Ein entsprechendes Assemblerprogramm erfordert da schon etwas mehr Aufwand, da PRINT und GET keineswegs Teil des 6502-Befehlssatzes sind.

Ein Menü braucht zwei grundlegende Funktionen: Da wäre die Ausgabe von Strings auf den Bildschirm, damit Überschrift und Menü-Punkte angezeigt werden können. Weiterhin braucht man die Möglichkeit, ein Zeichen von der Tastatur zu lesen, um zu wissen, welcher Menue-Punkt gewählt wurde.

Hilfe vom Betriebssystem

Bei beiden Funktionen greift uns das Betriebssystem des Atari-Computers unter die Arme. Es enthält eine zentrale Routine, die alle Ein- und Ausgaben erledigt: der Central-Input-Output (CIO). Damit dieses Programm auch weiß, was zu tun ist, muß man vor dessen Aufruf eine Tabelle, den IOCB (Input-Output-Control-Block), ausfüllen. Dort wird eingetragen, welche Operation gewünscht wird, wo die betreffenden Bytes stehen, und wieviele davon bearbeitet werden sollen.

Der Grundgedanke von CIO ist eine Vereinfachung der vielfältigen Ein- und Ausgaben, die in einem Computer anfallen. Dazu werden alle Quellen oder Senken (Ziele) von Daten als 'Geräte' betrachtet, die jeweils mit einem Buchstaben und einem Doppelpunkt gekennzeichnet sind. Es spielt dabei keine Rolle, ob es sich um tatsächliche Geräte wie Floppy (D:) oder Drucker (P:) oder um reine 'Software-Geräte' wie z.B. Bildschirmeditor (E:) oder etwa eine RAM-Disk handelt. CIO stellt einen Satz von Befehlen bereit, der auf jedes Gerät angewendet werden kann. Das bedeutet, wenn Sie die Ausgabe von Texten auf den Bildschirm beherrschen, so können Sie das selbe Schema anwenden um Texte auf Drucker, Floppy, Modem etc.

auszugeben. Natürlich erfordern manche Geräte zusätzliche Informationen (z.B. die Floppy einen Filenamen), aber das Prinzip ist immer gleich.

Wenn Sie die Philosophie von CIO verstanden haben, können Sie praktisch alle Ein-/Ausgabe Operationen auch in Maschinensprache vornehmen. Daher soll auch vor der weiteren Besprechnung des Menüprogrammes erst ein Blick auf diesen wichtigen Teil des Betriebssystems geworfen werden.

2.1.1 CIO und das Betriebssystem

Was steckt eigentlich hinter dem Namen 'Betriebssystem'? Nichts weiter als ein ziemlich langes Maschinenprogramm (ca. 13KByte), das eine Sammlung von Routinen enthält, die man zum 'Betrieb' des Computers häufig benötigt. Es handelt sich um Routinen zur Ein-/Ausgabe, zur Verwaltung der Interrupts und zur Vorbereitung des Computers nach dem Einschalten. Mit dem Betriebssystem wird erreicht, daß alle Programme einen gewissen Grundstock von elementaren Funktionen zur Verfügung haben. Damit wird vermieden, daß der Programmierer gewissermaßen das Rad immer auf's Neue erfinden müßte. Es ist daher ziemlich nützlich, wenn man sich im Betriebssytem des Atari-Computers etwas auskennt. Es genügt, wenn man eine wesentliche Einsprungadressen kennt und weiß, wie die Daten zu übergeben sind. Alle Einsprungadressen liegen im Adressbereich $E400 bis $E48F, dort befindet sich eine Sprungtabelle (nur JMP-Befehle zu den eigentlichen Routinen).

WICHTIG:

> Verwenden Sie niemals andere Einsprünge, sonst laufen Sie Gefahr, daß ihre Programme auf älteren oder zukünftigen Versionen von Atari-Computern nicht funktionieren. Nur die oben genannten Adressen sind von den Entwicklern des Betriebssystemes garantiert, und sind auch angefangen von der Serie 400/800 bis hin zum 130XE gleich geblieben.

Ein Teil des Betriebssystemes ist auch die CIO-Routine. Sie wird durch einen ganz normalen Unterprogrammsprung auf die Adresse $E456 aktiviert, doch muß zuvor ein IOCB ausgefüllt werden. IOCBs gibt es insgesamt 8 Stück, das hat den Hintergrund, daß auch mehrere Kanäle zur Ein-/Augabe (Files) zugleich offen sein können. Sie kennen dies von OPEN und CLOSE in BASIC: dort muß auch eine Nummer von 0 bis 7 angegeben werden. Mit dieser Nummer wählen Sie einen der 8 IOCBs aus.

IOCBs

Ein IOCB besteht aus 16 Bytes, die in jedem IOCB die gleiche Struktur haben. Die Aufgaben der einzelnen Bytes können Sie der Tabelle 2.1-1 entnehmen. Wichtigstes Byte

Name	Offset	Länge	Beschreibung
ICHID	0	1	Index in Geräte-Tabelle, wird vom Betriebssystem (OS) gesetzt
ICDNO	1	1	Gerätenummer (z.B aus D2:) wird vom OS gesetzt
ICCOM	2	1	Hier muß der Code des gewünschten Befehles eingetragen werden.
ICSTA	3	1	Gibt den Status des letzten Kommandos an. (Fehlercode, der auch im Y-Register zu finden ist). wird vom OS gesetzt.
ICBAL ICBAH	4	2	Buffer-Adresse. Gibt die Adresse an, ab der das CI0-Kommando arbeiten soll.
ICPUT	6	2	Zeiger auf die Adresse der PUT-BYTE Routine des Gerätetreibers minus 1. Wird vom OS gesetzt, nur für interne Zwecke!
ICBLL ICBLH	8	2	Bufferlänge: Länge des zu bearbeitenden Bereiches in Bytes. Dieser Wert wird pro bearbeitetem Byte um eins vermindert.
ICAX1	10	1	Zusatzbyte Nr. 1. Dient beim OPEN- zur Angabe der Datenflußrichtung.
ICAX2	11	1	Zusatz-Byte Nr. 2. Wird von einigen Gerätetreibern benutzt.
ICAX3 ICAX4 ICAX5	12 13 14	1 1 1	Zusatz-Bytes 3-5: werden im Zusammenhang mit NOTE und POINT benutzt.
ICAX6	15	1	Zusatz-Byte 5: wird derzeit nicht benutzt.

Tabelle 2.1-1: Struktur eines IOCBs

ist ICCOM, in dem ein Code für die gewünschte Funktion eingetragen wird. Hier die wichtigsten Codes (mit den gewöhnlich verwendeten symbolischen Namen):

Funktion	CODE	Name	IOCB-Einträge
OPEN	3	COPN	*Filename,Modus in ICAX1
CLOSE	12	CCLOSE	(kein weiterer Eintrag)
Text-Input	5	CGTXTR	*Buffer, Länge
Text-Print	9	CPTXTR	*Buffer, Länge
BGET	7	CGBINR	*Buffer, Länge
BPUT	11	CPBINR	*Buffer, Länge

In den meisten Fällen müssen neben ICCOM weitere Felder mit zusätzlichen Angaben gefüllt werden, dies ist in der letzten Spalte eingetragen. Ein Stern vor dem Namen bedeutet dabei, das nur ein Zeiger (Adresse) eingetragen werden muß. Dazu gleich ein Beispiel: Nehmen wir an, Sie wollen einen Kanal zur Ausgabe auf den Drucker öffen. In BASIC wäre das mit einem OPEN-Befehl zu erledigen:

```
OPEN #2,8,0,"P:"
```

In Maschinensprache sind da schon ein paar mehr Befehle nötig:

```
OPEN    LDX #$20
        LDA #COPN
        STA ICCOM,X
        LDA #FNAME:L
        STA ICBAL,X
        LDA #FNAME:H
        STA ICBAH,X
        LDA #8
        STA ICAX1,X
        LDA #0
        STA ICAX2,X
        JSR CIOV
        BMI FEHLER
...

FNAME   ASC "P:"
```

Von großer Wichtigkeit ist der erste LDX-Befehl, denn über das X-Register wird CIO mitgeteilt, welcher der acht IOCBs bearbeitet werden soll. X wird mit der IOCB-Nummer mal 16 geladen, wenn also wie im Beispiel IOCB Nr. 2 gemeint ist, muß X gleich 32 ($20) sein. Es hat sich in der Praxis bewährt, das X-Register immer gleich zu Beginn einer E/A-Operation mit dem IOCB-Zeiger zu laden, denn dann kann man die einzelnen Felder des IOCBs per X-Indizierung direkt ansprechen.

Mit dieser Art des Zugriffes braucht man im DefinitionsTeil des Listings nur den Adressen des IOCBs Nr. 0 symbolische Namen zuzuordnen. Da jeder IOCB genau 16 Bytes lang ist, sind jeweils entsprechende Einträge in 16 Bytes Abstand zu finden, genau der Grund, warum die IOCB-Nummer multipliziert mit 16 im X-Register abgelegt werden muß.

Bei OPEN wird anfangs der Befehlscode COPN (3) eingetragen, dann folgt ein Zeiger auf den Filenamen "P:", der weiter unten im Programm als Zeichenkette definiert ist. Beachten Sie dabei das EOL-Zeichen am Ende des Strings! Schließlich werden die Hilfs-Felder ICAX1 und ICAX2 ausgefüllt. ICAX1 hat dabei die Aufgabe, die Richtung des Datenflußes festzulegen (4=Eingabe, 8=Ausgabe, 12=Ein- & Ausgabe u.a.).

Fehlerbehandlung

Nachdem die Eingaben für CIO festgelegt und das X-Register mit dem IOCB-Offset geladen ist, wird CIO aufgerufen. CIO führt die befohlene Arbeit aus und gibt anschließend im Y-Register einen Status-Code zurück, der über das Auftreten von Fehlern informiert.

Ist z.B. der IOCB schon in Benutzung, so wird im Y-Register der Fehlercode $81 (129) gemeldet, die Codes sind übrigens identisch mit den Fehlermeldungen in BASIC ab Nummer 128. Als weitere Hilfe zur schnellen Abfrage wird im Fehlerfall das N-Flag gesetzt, und ein BMI kann entscheiden, ob alles gut ging oder ob zu einer Fehler-Routine verzweigt werden muß. Anhand des Codes im Y-Register kann diese Routine den Grund des Fehlers herausfinden und z.B. eine Meldung an den Benutzer abgeben.

> Grundsätzlich gilt: Ist das N-Flag gesetzt und damit der Code im Y-Register größer gleich 128, dann ist ein Fehler aufgetreten.

Ein Tip am Rande: Vor einem OPEN immer zuerst einen CLOSE Befehl auf

den IOCB ausgeben. Damit erspart man sich den lästigen Fehler 129, der auftritt wenn der IOCB zuvor nicht ordnungsgemäß geschlossen wurde.

2.1.2 Menü-Programm intern

Zurück zum Menü-Programm. Mit den eben erworbenen Kenntnissen ist das anfangs beschriebene Vorhaben kein Problem mehr. Das Listing beginnt mit den Definitionen der IOCB-Labels, CIO-Befehlen und einigen sonstigen Adressen und Konstanten. Der Objektcode wird ab Adresse $A800 abgelegt, Sie wissen ja, das ist der geschützte Bereich für Maschinenprogramme beim ATMAS-II.

Bei der Besprechung des Programmes beginnen wir am besten mit den Unterprogrammen. Da ist zunächst die Routine PRINT, die dafür sorgt, daß ein String auf den Bildschirm ausgegeben wird. Dem Unterprogramm wird die Anfangsadresse des zu druckenden Strings in zwei Registern übergeben, das LSB im Akku, das MSB im Y-Register. Zur Ausgabe wird die CIOFunktion 'Text-Print' (CPTXTR) verwendet. Wie aus der obigen Tabelle ersichtlich, muß dazu die String-Adresse in den IOCB eingetragen werden, außerdem muß eine maximale Länge angegeben werden (hier willkürlich 127). CIO erkennt das Ende eines Strings am EOL-Zeichen (Code $9B), daher finden Sie im Listing auch nach jedem String eine Zeile mit "DFB EOL"

Sie werden sich wahrscheinlich wundern, warum hier kein Kanal zur Ausgabe geöffnet werden mußte. Ganz einfach: Der Kanal zur Bildschirmausgabe wird beim Einschalten des Computers automatisch auf den IOCB#0 geöffnet. Er kann später beliebig zu Ein- und Ausgaben am Bildschirm herangezogen werden.

Tastatur

Etwas aufwendiger ist die Abfrage der Tastatur. Natürlich könnten wir einfach wieder der IOCB#0 mit einer Text-Input Funktion verwenden. Das Ergebnis wäre mit einem INPUT in BASIC vergleichbar. Was wir aber wollen ist aber mehr die Simulation des GET-Befehles, sonst müßte nach der Eingabe noch RETURN gedrückt werden. Nebenbei bemerkt: Wie ein INPUT mit CIO programmiert wird, können Sie sich in IOLIB auf der ATMAS-II Systemdiskette ansehen. Was ist nun nötig, um ein Zeichen von der Tastatur zu bekommen? In BASIC müßten wir so vorgehen:

```
OPEN #2,8,0,"K:":GET#2,A:CLOSE#2
```

Damit ist die Routine HOLZEI (Hole Zeichen) eigentlich schon vorgegeben. Zuerst den IOCB für ein OPEN ausfüllen (s. o.), dann ein GET per CIO ausführen und dann den Kanal mit CLOSE wieder schließen. Beim OPEN wird der Trick mit CLOSE benutzt, um den Fehler 129 zu verhindern. Das GET ist für CIO ein Spezialfall des BGET-Befehles. Wenn Ihnen das kein Begriff ist - kein Wunder, den er wird in BASIC auch nicht verwendet. Mit BGET kann man einen ganzen Speicherbereich auf einmal einlesen, das ist sehr nützlich, wenn man z.B. Graphik-Bilder oder Zeichensätze von der Diskette einlesen möchte (s. /1/ Seite 66). Auf das Eingabegerät Tastatur bezogen gibt das natürlich nicht viel Sinn, denn dort will man meist nur eine Taste auslesen und diese sofort verarbeiten. Dazu verwendet man ebenfalls den BGET-Befehl von CIO, nur setzt man die Buffer-Länge (ICBLL und ICBLH des IOCBs) auf Null. Damit haben wir CIO mitgeteilt, daß wir nur ein einziges Byte haben wollen, und CIO gibt dieses Byte im Akku zurück.

Hauptprogramm

Das Hauptprogramm (ab Label MENUE) benutzt mehrmals die PRINT-Routine, um den Bildschirmaufbau für das Menü zu erzeugen. Dabei werden COLCRS und ROWCRS benutzt, das sind spezielle Speicherzellen des Betriebssystemes, die die Cursor-Position enthalten. Verändert man Sie, kann man den Cursor beliebig bewegen (entspricht POSITION in BASIC).

Nun wird in einer Schleife die Tastatur mit HOLZEI abgefragt und das Ergebnis mit CMP-Befehlen geprüft. Ergibt sich eine Übereinstimmung, so wird das entsprechende Unterprogramm aufgerufen, ansonst auf das nächste Zeichen gewartet. Als Demos für die Funktionen wurden Veränderungen der Farben gewählt, Sie können aber selbstverständlich die Unterprogramme nach eigenen Vorstellungen ausbauen. Interessant ist vielleicht noch der Trick bei FUNKT3, der es ermöglicht, aus dem Unterprogramm in den Monitor zurückzukehren. Es wird mit zwei PLA-Befehlen die letzte Rücksprungadresse (die von JSR FUNKT3) vom Stack genommen, damit bleibt nur noch die vom Aufruf des gesamten Programmes. Ein RTS kehrt damit zum Monitor von ATMAS-II zurück. Klar, das entspricht nicht gerade den Vorstellungen vom strukturierten Programmieren, aber so etwas werden Sie in Assemblerprogrammen häufig finden. Ein wichtiges Kapitel bei der Programmierung von Ein-/Ausgaben ist die Behandlung von Fehlern. Im Beispiel wurde das etwas salopp behandelt, um das Programm nicht unnötig aufzublähen. Tritt ein

Fehler auf, so wird mit BRK der Monitor aufgerufen und Sie sehen die Fehlernummer im Y-Register. Das dürfen Sie natürlich nicht machen, wenn Sie einmal ein richtiges Anwenderprogramm schreiben sollten. Dann müssen Sie hier weitaus aufwendigere Programmteile zur Meldung des Fehlers vorsehen.

```
**********************************
*         MENUEPROGRAMM   V1.20
*
*         P. Finzel       1986
**********************************
*
* Definitionen des IOCBs Nr. 0 *
*
ICCOM     EQU $342        Kommando-Byte
ICSTA     EQU $343        Status-Byte
ICBAL     EQU $344        Buffer-Adresse
ICBAH     EQU $345
ICBLL     EQU $348        Buffer-Laenge
ICBLH     EQU $349
ICAX1     EQU $34A        Zusatz-Byte 1
ICAX2     EQU $34B        Zusatz-Byte 2
*
*         CIO-Befehle
*
COPEN     EQU 3           Datei oeffnen
CCLOSE    EQU 12          Datei schliessen
CSTXT     EQU 5           'Set Text' (INPUT)
CPTXT     EQU 9           'Put Text' (PRINT)
CGBIN     EQU 7           'Set Binary' (BGET)
CPBIN     EQU 11          'Put Binary' (BPUT)
*
CIOV      EQU $E456       CIO-Einsprung
*
* OS-Register & Sonstiges
*
EOL       EQU $9B         RETURN-Zeichen
CLS       EQU $7D         Zeichen f. 'Clear Screen'
ROWCRS    EQU $54         Cursor-Zeile
COLCRS    EQU $55         Cursor-Spalte
COLOR1    EQU 709         Farbe der Zeichen
COLOR2    EQU 710         Farbe Text-Hintergrund
SPALTE    EQU 10          linker Rand
```

```
*
* PROGRAMMSTART
*
          ORG $A800

MENUE     LDA #TITEL:L    Ueberschrift
          LDY #TITEL:H    ausdrucken
          JSR PRINT

          LDA #SPALTE     POSITION
          STA COLCRS      ausfuehren
          LDA #7
          STA ROWCRS
          LDA #WAHL1:L    erste Menue-Zeile
          LDY #WAHL1:H    ausgeben
          JSR PRINT

          INC ROWCRS      fuer Leerzeile
          LDA #SPALTE     POSTION fuer
          STA COLCRS      zweite Zeile
          LDA #WAHL2:L    zweiten String
          LDY #WAHL2:H    drucken
          JSR PRINT

          INC ROWCRS      POSTION fuer
          LDA #SPALTE     dritte Zeile
          STA COLCRS
          LDA #WAHL3:L    und Zeile drucken
          LDY #WAHL3:H
          JSR PRINT

          LDA #18         POSITION Zeile 18
          STA ROWCRS      Spalte 5
          LDA #5
          STA COLCRS
          LDA #FRAGE:L    Text 'Bitte ...'
          LDY #FRAGE:H    ausdrucken
          JSR PRINT

*
* Zeichen von Tastatur holen,
* und Bereich pruefen (1 bis 3)
*
SCHLF     JSR HOLZEI      Zeichen in AKKU
          CMP #'1         ist eine Eins?
          BNE IST2        nein -->
          JSR FUNKT1      Fkt. ausfuehren
          JMP ENDE
IST2      CMP #'2         eine Zwei?
          BNE IST3        nein -->
          JSR FUNKT2      Fkt. ausfuehren
          JMP ENDE
IST3      CMP #'3         etwa eine 3?
          BNE ENDE        Eingabe ungueltig->
          JSR FUNKT3      dritte Fkt.
ENDE      JMP SCHLF
```

Menüprogramm (Fortsetzung)

```
*
* Strings fuer Texte und OPEN
*
KEYBRD    ASC "K:"
          DFB EOL
TITEL     DFB CLS
          ASC "DEMO EINES MENUEPROGRAMMES"
          DFB EOL
WAHL1     ASC "1 - Weiss"
          DFB EOL
WAHL2     ASC "2 - Schwarz"
          DFB EOL
WAHL3     ASC "3 - Ende"
          DFB EOL
FRAGE     ASC "Bitte Nummer tippen!"
          DFB EOL

*****************************************
* ZEICHEN VON TASTATUR LESEN
*
* IOCB#2 oeffnen, Zeichen lesen und
*        Kanal wieder schliessen
* AUSGABE: <A>: Zeichen im ASCII-Code
*****************************************

HOLZEI    LDX #$20        zuerst den
          LDA CCLOSE      IOCB vorsichtshalber
          STA ICCOM,X     schliessen
          JSR CIOV        (s. Text!)

          LDX #$20
          LDA #COPEN      jetzt den
          STA ICCOM,X     OPEN-Befehl
          LDA #KEYBRD:L   mit Filenamen
          STA ICBAL,X     "K:" geben
          LDA #KEYBRD:H
          STA ICBAH,X
          LDA #4          Die Hilfs-Bytes
          STA ICAX1,X     auf Eingabe
          LDA #0          stellen
          STA ICAX2,X
          JSR CIOV        und Zeichen lesen
          BMI IOERR       Fehler?
*
          LDX #$20        IOCB Nr.2
          LDA #0
          STA ICBLL,X     Bufferlaenge
          STA ICBLH,X     gleich Null
          LDA #CGBIN      Befehl Zeichen
          STA ICCOM,X     holen
          JSR CIOV        Zeichen in AKKU
          BMI IOERR       Fehler-->
```

Menüprogramm (Fortsetzung)

```
          PHA             Zeichen retten
          LDX #$20        Kanal wieder
          LDA #CCLOSE     schliessen
          STA ICCOM,X     (CLOSE)
          JSR CIOV        kein Fehler
          PLA             Zeichen zurueck
          RTS             moeglich!

**********************************
* Unterprogramm zum Ausdruck eines
* Strings: LSB in <A>
* MSB in <Y>
**********************************

PRINT     LDX #0          IOCB#0
          STA ICBAL,X     zeiger auf
          TYA             String eintragen
          STA ICBAH,X     MSB
          LDA #127        max. Laenge
          STA ICBLL,X     (gewaehlt 127)
          LDA #0          MSB der Laenge
          STA ICBLH,X     ist null
          LDA #CPTXT      Befehl 'Put Text'
          STA ICCOM,X
          JSR CIOV        ausgeben ==>
          BMI IOERR       Fehler?? ->
          RTS

IOERR     BRK             FEHLER!!

**********************************
* Ab hier die Unterprogramme der
* einzelnen Funktionen
**********************************

FUNKT1    LDA #12         Hintergr. hell
          STA COLOR2
          LDA #0          Schrift dunkel
          STA COLOR1
          RTS

FUNKT2    LDA #0          Hintergr. dunkel
          STA COLOR2
          LDA #12         Schrift hell
          STA COLOR1
          RTS

FUNKT3    PLA             Ruecksprungadresse
          PLA             vom Stack holen
          RTS             zurueck zum Monitor!
```

Menüprogramm (Schluß)

2.2 Programmieren der Player-Missile Graphik

Dieser Abschnitt ist dem Umgang mit den speziellen Fähigkeiten des ATARI-Computers gewidmet. Wie Sie wissen, gibt es im I/O-Bereich zahlreiche 'Hardware-Register' mit denen man ein wahres Feuerwerk an Grafiken, Farben und Bewegungen auslösen kann. Speziell soll hier ein Beispiel zur Programmierung der Player-Missile Graphik erläutert werden und Ihnen den Umgang mit den Hardware-Registern näher bringen.

2.2.1 Hardware- und Schattenregister

Jeder der vier Peripherie-Bausteine hat im I/O-Bereich eine Speicherseite (eine Page) für seine Register reserviert:

D000-D0FF: GTIA,	hier finden Sie Register für Farben und für die PM-Graphik.
D200-D2FF: POKEY	enthält die Register für Sound IRQ-Interrupts und Datenübertragung am seriellen Bus.
D300-D3FF: PIA	ist ein einfacher Baustein zur parallelen Ein-/Ausgabe und wird beim Atari hauptsächlich für Joysticks und Bank-Switching eingesetzt.
D400-D4FF: ANTIC	heißt der Baustein zur Steuerung der Bildschirm-DMA und der NMI-Interrupts.

Keiner dieser vier elektonischen Spezialbausteine nützt alle 256 möglichen Adressen seiner Speicherseite voll aus, sie begnügen sich in der Regel mit vier (PIA) bis 32 (GTIA) Adressen, eben je nach Komplexität des Bausteines.

Viele dieser Register können entweder nur gelesen oder nur beschrieben werden, andere haben gar beim Lesen und beim Schreiben ganz andere Funktionen. So wird z. B. mit einem Schreibbefehl auf die Adresse $D200 die Frequenz des ersten Sound-Kanales festgelegt, beim Lesen dieser Adresse wird hingegen der Wert des ersten Drehreglers erfaßt.

Schattenregister

Nicht nur aus diesem Grund gibt es für einige der Hardware-Register zusätzliche 'Schattenregister'. Hinter diesem Begriff verbergen sich normale RAM-Speicherzellen, die vom Betriebssystem regelmäßig in die Hardware-Register übertragen werden. Dieser Vorgang geschieht 50 mal in der Sekunde mit Hilfe eines Interrupts (des VBIs, mehr darüber lesen Sie im nächsten Abschnitt).

Der Vorteil von solchen Schattenregistern liegt klar auf der Hand: Sie können in jedem Fall gelesen und beschrieben werden. Mit anderen Worten heißt das, daß man einen dort hineingeschriebenen Wert auch wieder in Erfahrung bringen kann. Wenn Sie z. B. einen Wert per STA-Befehl in das Hardware-Register für die Bildschirmfarbe schreiben würden, dann wäre es unmöglich den Wert mit einem LDA wieder zu lesen, geschweige denn die Farbe durch INC-Befehle zu erhöhen. Man kann das aber wohl mit dem Schattenregister tun. Da die Übertragung der Schatten- in die Hardware-Register zu einem Zeitpunkt stattfindet, in dem kein Video-Bild erzeugt wird, kann es bei der Änderung von Schattenregistern auch nie zu Störungen der Graphik kommen. Schattenregister gibt es für die Farbregister, die Joysticks, Drehregler und einige wichtige Register zur Steuerung der Spezialbausteine (DMA, Display-List und Interrupts).

2.2.2 PM-Graphik

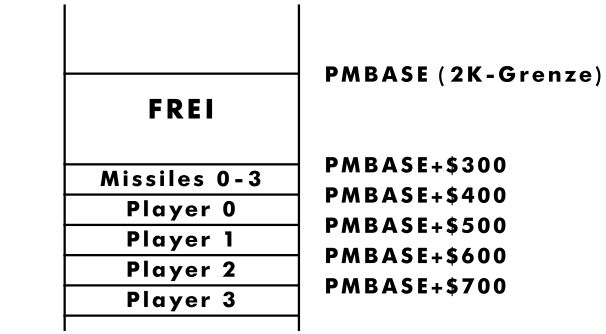
Für alle, die sich noch nicht mit PM-Graphik beschäftigt haben erst ein paar Worte zur Funktion. Mit PM-Graphik kann man bis zu acht Objekte darstellen, die sich unabhängig von der Hintergrundgraphik bewegen lassen. Die Objekte kann man sich als durchsichtige vertikale Streifen vorstellen, die nur dort sichtbar werden, wo ein Bit im PM-Speicher gesetzt ist. Jeder der vier Players ist 8 Pixels breit, daher wird jede Zeile eines Players durch ein Byte dargestellt. Die gesamte Auflösung eines Players beträgt bei einzeiliger Darstellung 256 (vertikal) mal 8 Punkte (horizontal), wobei sich in vertikaler Richtung oben und unten 32 Pixels außerhalb des regulären Bildschirmes befinden. Die Bewegung eines Objektes in horizontaler Richtung erfolgt durch Verschieben des 'Streifens' mit Hilfe von Positions-

Registern, in vertikaler Richtung muß das Bitmuster des Objektes per Software innerhalb des PM-Speichers verschoben werden. Der letztere Vorgang läßt sich recht leicht bewältigen, da sich die 256 Bytes jedes Players in einer Page befinden. Die Missiles haben je eine maximale Auflösung von 256 mal 2, alle vier sind im Speicher in einer Page zusammengefaßt.

Kooperation

Die PM-Graphik ist ein Ergebnis der Zusammenarbeit von ANTIC und GTIA. Während ANTIC die Daten ohne Mithilfe des 6502-Prozessors aus dem Speicher holt (man spricht hier von DMA), sorgt GTIA für die Umsetzung der Daten in Farben und Formen. Da zwei Bausteine programmiert werden müssen, gilt es, eine ganze Reihe von Hardware- bzw. deren Schattenregister zu bedienen.

Es würde den Rahmen dieses Buches übersteigen, wenn sämtliche Details der PM-Graphik besprochen würden. Aus diesem Grund soll hier nur Eingang finden, was für das nachfolgende Demo-Programm Bedeutung hat.



PM-Speicher bei einzeiliger Auflösung

Als erstes muß für die PM-Graphik ein Speicherbereich reserviert werden, ganz ähnlich, wie das für den normalen Videospeicher erfolgen muß. Da wir die 'einzeilige' Auflösung benutzen wollen, sind 2 KByte erforderlich. Der Anfang dieses Bereiches darf nicht beliebig gewählt werden, sondern muß an einer Adresse beginnen, die ein Vielfaches von 2048 ($800) ist. Analog zur 'Page-Grenze' spricht man hier auch von einer '2K-Grenze'. Nur der höherwertige Teil

dieser Anfangsadresse wird in das Register PMBASE geschrieben, so daß ANTIC weiß, woher seine DMA die Daten holen soll.

Jetzt wird die Player-DMA im Schattenregister SDMCTL ($22F) eingeschaltet, somit beginnt ANTIC bereits die Daten aus dem Speicher zu holen. Nun muß noch GTIA programmiert werden, damit die Objekte auch am Bildschirm sichtbar werden. Dies geschieht mit dem Register GRACTL ($D01D). Mit GPRIOR ($26F) wird festgelegt, ob bei Überlappungen die PMGraphik oder die Hintergrundgraphik den Vorrang hat. Folgende Register sind für jeden Player und einmal für die Missiles vorhanden:

PCOLR0 ($02C0):	Farbe von Player 0 (Farbe*16 + Helligkeit)
SIZEP0 ($D008):	Breite von Player 0 (0:einfach, 1: doppelt 3:vierfach)
GRAFP0 ($D000):	Umriß des Players 0 (wird durch DMA bedient!)
HPOSP0 ($D000):	Horizontale Position von Player 0 (0-255, 48 bis 208 sichtbar)

Noch ein paar Worte zu den GRAFPx-Registern: Hier kann der 'Umriß' des Players (oder der Missiles) eingetragen werden. Das funktioniert aber nur, wenn die ANTIC-DMA nicht benutzt wird. Mit anderen Worten: Man kann die Players auch erzeugen, indem der 6502 zu Beginn jeder Bildschirmzeile den Umriß des Players in das entsprechende GRAFPx-Register schreibt. Diese Methode erfordert einen gewaltigen Aufwand an Timing, da man den 6502 absolut an den Ablauf des Elektronenstrahles koppeln muß. Bei dieser Aufgabe ist selbst die enorme Geschwindigkeit von Assemblerprogrammen schon kritisch!

2.2.3 Beispielprogramm zur PM-Graphik

Als Demo zur PM-Graphik wollen wir ein praxisnahes Beispiel heranziehen. Ein Player wird als Cursor verwendet und kann per Joystick über den Bildschirm bewegt werden. Wie gewohnt brauchen sie nur das nachfolgende Listing eintippen und assemblieren. Sie können es dann an der Adresse $A800 mit der GO-Funktion des Monitors starten. Stecken Sie dann einen Joystick in die Buchse 1 und probieren Sie das Programm aus.

Definitionsteil

Hardware- und Schattenregister werden am Anfang des Programmes per EQU-Definitionen mit symbolischen Namen belegt, danach werden zwei Variable XPOS und YPOS eingeführt, die später die Koordinaten des Cursors festhalten sollen. In der nachfolgenden 10 Byte langen Tabelle SHAPE wird die Form des Cursors festgelegt. Jede Eins in den Binärzahlen steht für einen sichtbaren Punkt, die Nullen sind quasi 'durchsichtig'. Wichtig ist hier auch, daß die Tabelle sowohl oben als auch unten von einer Leerzeile (Nullen) begrenzt wird. Wenn später das Muster um eine Position versetzt in den Player-Speicher kopiert wird, sorgt diese Null dafür, daß keine Reste des alten Musters zurückbleiben.

Das Programm selbst wird in den geschützten Bereich ab $A800 assembliert, der PM-Speicher beginnt ab Adresse $5000. Dort liegt zwar die Symboltabelle von ATMAS-II, diese wird jedoch nach dem Assemblieren nicht mehr gebraucht und kann ohne Bedenken überschrieben werden.

Das Programm

Die Variablen und Shape-Definitionen werden durch einen JMP-Befehl zum Label START übersprungen. Nun wird zunächst der gesamte PM-Speicherbereich mit einer kleinen Schleife gelöscht, so daß nach dem Einschalten der PM-Graphik keine wirren Muster am Schirm entstehen. Dann werden die Register von ANTIC und GTIA gemäß den in 2.2.1 beschriebenen Regeln gesetzt und die PM-Graphik somit freigegeben.

In der Hauptschleife wird ein Unterprogramm JOYSTK aufgerufen, das die Koordinaten je nach Stellung des Joysticks verändert. Mit diesen neuen Werten kommt das

Unterprogramm PMSHAPE zum Zug, das die Form des Cursors aus der Tabelle SHAPE ab der durch die Y-Position (in YPOS) gegebene Stelle in den Speicherbereich für Player 1 kopiert. Durch diese Maßnahme erscheint der Cursor an der gewünschten vertikalen Position, die X-Position wird eingestellt, indem XPOS in das Hardware-Register HPOSP0 eingetragen wird. Am Ende jedes Durchlaufes wird geprüft, ob die Start-Taste gedrückt wurde, wenn ja, wird die Schleife verlassen.

In diesem Fall wird die PM-Graphik abgeschaltet. Man sollte dabei nicht vergessen, außer GTIA auch die ANTIC-DMA abzuschalten, da diese ein wenig Rechenzeit in Anspruch nimmt. Weiterhin empfiehlt es sich, die GRAFPx-Register zu löschen, sonst können unerwünschte Störungen am Bildschirm zurückbleiben. Wenn Sie wollen können Sie auch noch die Horizontal-Register auf Null setzen und so die Player für alle Fälle außer Sicht bringen.

Joystickabfrage

Im Unterprogramm JOYSTK können Sie sehen, wie man eine Abfrage eines Joysticks zweckmäßigerweise programmiert Aus dem Inhalt des Schattenregisters wird jeweils ein einziges Richtungs-Bit mit AND maskiert. Ist dieses Bit gesetzt (d.h. der Joystick NICHT in diese Richtung gedrückt), ist der Akku ungleich Null und BNE verzweigt.

Wird dagegen eine Richtung erkannt, so muß vor der Veränderung der Koordinaten überprüft werden, ob sich der Cursor schon am Rand des Bildschirmes befindet. Da die X-und Y-Koordinaten bereits zuvor in X- und Y-Register geladen wurden, kann dies mit CPX bzw. CPY erfolgen. Liegt keine Randpostion vor, so werden die Register mit Increment bzw. Decrement-Befehle gemäß der Richtung verändert.

Dieser Vorgang wird für alle vier Richungen unabhängig wiederholt, somit sind auch schräge Bewegungen möglich. Probieren Sie es aus!

Kopieren des Player-Shapes

Wie schon am Anfang dieses Abschnittes erwähnt, muß die vertikale Bewegung des Objektes durch Verschiebung der Daten im PM-Speicher erfolgen. Diese Aufgabe übernimmt das Unterprogramm PMSHAPE, allerdings wird das Shape hier nicht verschoben, sondern immer von der Shapetabelle an die neue Y-Koordinate kopiert. Da immer nur Bewegungen um einen

Schritt erlaubt sind, und das Shape oben und unten eine zusätzliche Leerzeile hat, wird das alte Shape sicher gelöscht, es bleiben keine unschönen 'Reste' zurück. Für die horizontale Verschiebung wird XPOS in das Register HPOSP0 übertragen.

Interessant sind auch die drei Befehle am Anfang dieses Unterprogrammes. Sie stellen sicher, daß der Cursor pro Bildschirmaufbau nur einmal bewegt wird und verhindern daher Störungen der Graphik. Auf der anderen Seite wird dadurch die Geschwingigkeit erheblich herabgesetzt. Löschen Sie die drei Befehle heraus (Vorsicht mit dem Label!) und probieren Sie das Programm nochmal. Sie werden Ihren Augen nicht trauen, wie schnell Assembler in Verbindung mit den speziellen Fähigkeiten des Atari-Computers sein kann.

Wenn Sie gerne Spiele programmieren möchten, dann können Sie dieses Programm als Grundgerüst für die Player-Bewegung hernehmen. Natürlich müssen dazu Erweiterungen zur Animation der Objekte etc. erfolgen. Außerdem empfiehlt es sich, die Bewegungs-Routinen im VBI ablaufen zu lassen (s. nächsten Abschnitt). Andere Anwendungen für diese Routinen: als Cursor für ein Zeichenprogramm, als Zeiger für eine 'Joystick'-Maus usw.

```
*********************************
*     DEMO ZUR BENUTZUNG DER
*       HARDWARE-REGISTER   V1.01
*
*         am Beispiel der
*    PLAYER-MISSILE GRAPHIK
*
* P. Finzel                 1986
*********************************

HPOSP0     EQU $D000       Hor.-Position
SIZEP0     EQU $D008       Breite der Player
GRAFP0     EQU $D00D       Umriss
GRACTL     EQU $D01D       Graphik-Kontrollreg.
PMBASE     EQU $D407       PM-Basisadresse
CONSOL     EQU $D01F       Funktionstasten

RTCLOK     EQU $12         Uhr (3 Bytes)
SDMCTL     EQU $22F        DMA-Kontrollreg.
GPRIOR     EQU $26F        Prioritaeten
STICK0     EQU $278        Joystick 0
PCOLR0     EQU $2C0        Farbe Players

ADRPM      EQU $5000 PM-Bereich

           ORG $AB00

           JMP START Daten ueberspringen
```

```
XPOS      DFB 124          Variablen fuer
YPOS      DFB 124          Cursor-Position
*
* Bitmuster fuer Cursor-Shape
*
SHAPE     DFB %00000000
          DFB %00011000
          DFB %00011000
          DFB %00011000
          DFB %11111111
          DFB %11111111
          DFB %00011000
          DFB %00011000
          DFB %00011000
          DFB %00000000

START     LDA #0           PM-Bereich
          LDX #0           loeschen
LOESCH    STA ADRPM+$300,X Missiles
          STA ADRPM+$400,X Player 1
          STA ADRPM+$500,X ...
          STA ADRPM+$600,X
          STA ADRPM+$700,X Player 4
          DEX
          BNE LOESCH
*
          LDA #ADRPM:H     PM-Basisadresse
          STA PMBASE       festlegen
          LDA #$0C         Player 1 ganz
          STA PCOLR0       in weiss
          LDA #0           kleinste Breite
          STA SIZEP0       waehlen
          LDA #1           Prioritaet
          STA GPRIOR
          LDA #$3A         Player-DMA ein-
          STA SDMCTL       schalten
          LDA #2           Player Darstellung
          STA GRACTL       einschalten
*
* Hauptschleife
*
HAUPT     JSR JOYSTK       Joystick abfragen
          JSR PMSHAPE      Shape kopieren
          LDA CONSOL       START gedrueckt?
          AND #1
          BNE HAUPT        nein -->
*
* PM abschalten
*
          LDA #0           Darstellung ab-
          STA GRACTL       schalten
          LDA #$22         dann DMA aus
          STA SDMCTL
          LDA #0           und den Spieler-
          LDX #3           umriss loeschen
PMAUS     STA GRAFP0,X
          DEX
```

```
          BPL PMAUS
          RTS             fertig!

* Unterpgm: Abfrage des Joysticks
*
* Ausgabe: neue Position in XPOS,YPOS
*
JOYSTK    LDX XPOS        bisherige Positionen
          LDY YPOS        laden
          LDA STICK0      Joystick 0
          AND #1          nach oben?
          BNE NOBEN       nein -->
          CPY #32         Rand erreicht?
          BCC NOBEN       ja! -->
          DEY             nach oben
NOBEN     LDA STICK0
          AND #2          nach unten?
          BNE NUNTEN      nein -->
          CPY #215        Rand?
          BCS NUNTEN      ja -->
          INY             Shape 'runter
NUNTEN    LDA STICK0
          AND #4          nach links?
          BNE NLINKS      nein -->
          CPX #49         Rand?
          BCC NLINKS      ja -->
          DEX             Shape links
NLINKS    LDA STICK0
          AND #8          nach rechts?
          BNE NRECHTS     nein -->
          CPX #200        rechter Rand?
          BCS NRECHTS     ja -->
          INX             Shape rechts
NRECHTS   STX XPOS        neue Werte
          STY YPOS        speichern
          RTS

*
*         Unterpgm: Shape in PM-Bereich kopieren
*
* Eingaben: XPOS, YPOS
*
PMSHAPE   LDA RTCLOK+2    warten, bis Uhr weiter-
PMWAIT    CMP RTCLOK+2    zaehlt (somit ist
          BEQ PMWAIT      Video-Bild fertig!)
          LDA XPOS        Horzontale Pos.
          STA HPOSP0      in Hardware-Reg.
          LDY YPOS        Shape in aktuelle
          LDX #0          vertikale Position
KOPIERE   LDA SHAPE,X     eintragen
          STA ADRPM+$400,Y
          INY             naechste Zeile
          INX
          CPX #10         alle Zeilen?
          BNE KOPIERE     nein -->
          RTS
```

2.3 Interrupts

Interrupts sind eine unglaublich vielseitige Fähigkeit des Atari-Computers, die man als Assemblerprogrammierer unbedingt beherrschen sollte. Besonders Graphik- oder Musik-Anwendungen wären ohne Interrupts nur sehr mühsam zu programmieren.

Nicht zuletzt kann man bei der Programmierung von Interrupts eine Menge lernen, da sehr spezielle Techniken der Assemblerprogrammierung eingesetzt werden müssen.

2.3.1 Was ist ein Interrupt ?

Wenn Sie bisher nur in BASIC programmiert haben, dann sind Sie mit Interrupts wahrscheinlich noch nicht in Berührung gekommen. In BASIC kann man diese Fähigkeit des 6502-Prozessors nicht verwenden, da es schlicht und einfach viel zu langsam ist. Interrupts sind und bleiben eine Domäne der Assemblerprogrammierung. Aber was kann man sich eigentlich unter einem Interrupt vorstellen? Dazu ein Beispiel aus dem Alltag:

Sie sitzen am Computer und tippen eifrig Ihr neuestes Programm ein. Und wie sollte es anders sein - das Telefon läutet. Sie verlassen den Computer, gehen ans Telefon und kehren früher oder später wieder an den Computer zurück. Sie tippen dann - hoffentlich - genau an der Stelle weiter, an der Sie vor der Störung aufgehört haben.

Bei einem Computer würde man so einen Vorgang als 'Interrupt' (engl. für 'Unterbrechung') bezeichnen. Während ein Programm bearbeitet wird, fordert ein Peripherie-Baustein einen Interrupt an, d.h. er wünscht, daß der Prozessor seine derzeitige Arbeit beiseite legt und ein anderes Programm durchläuft. Nach der Bearbeitung dieser Routine nimmt der 6502 das unterbrochene Programm wieder auf, ganz so, als wäre nichts geschehen. Natürlich muß dafür gesorgt werden, daß die Arbeitsbedingungen des unterbrochenen Programmes durch den Interrupt in keinster Weise verändert werden. Stellen Sie sich vor, das Hauptprogramm beginnt gerade eine Rechenaufgabe im Akku und es wird von einem Interrupt unterbrochen. Nehmen wir weiter an, daß die

Interrupt-Routine den Akku auch zum Rechnen benutzt. Zweifellos würde das bedeuten, daß die Rechnung des Hauptprogrammes ein falsches Ergebnis liefern würde, da das Zwischenergebnis im Akku vom Interrupt überschrieben würde.

Register-Rettung

Grundsätzlich muß man daher am Anfang einer Interrupt-Routine die Register 'retten', im Regelfall werden die Inhalte der Register dazu per PHA auf den Stack geschoben. Bevor die Interrupt-Routine verlassen wird, kehrt man den Vorgang um, und holt die Werte wieder vom Stack zurück. Prozessor-Flags und Rücksprungadresse merkt sich der 6502 automatisch (auch am Stack), so daß das unterbrochene Programm vom Interrupt quasi nichts bemerkt. Natürlich muß man auch aufpassen, daß das Interrupt-Programm keine Speicherzellen ändert, die von anderen Programmen benutzt werden.

In der Hardware des Atari-Computers gibt es spezielle elektrische Verbindungen zwischen dem 6502 und den Peripherie-Bausteinen POKEY, PIA und ANTIC, die für die Übertragung der Interrupt-Anforderung zuständig sind. Daneben gibt es in jedem dieser Bausteine ein Register, in dem ein Bit für die Herkunft des Interrupts gesetzt wird. Um beim obigen Vergleich zu bleiben - es könnte anstatt des Telefons ja auch die Türglocke läuten. Man braucht daher ein Möglichkeit zur Unterscheidung der verschiedenen 'Interrupt-Quellen'.

Interrupt contra Polling

Vielleicht fragen Sie sich jetzt, wozu man denn das alles praktisch einsetzen kann. Nun, auch in diesem Fall können wir den Vergleich mit dem Telefon heranziehen. Stellen Sie sich nur vor, Ihr Telefon hätte gar keine Klingel. Dann müßten Sie alle ein oder zwei Minuten zum Telefon eilen, den Hörer abheben und lauschen, ob jemand angerufen hat. Klar - da bliebe nicht mehr viel Zeit für andere Dinge.

Auch einem schnellen Mikroprozessor ergeht es in dieser Beziehung nicht viel besser. Wenn er dauernd mit dem Abfragen irgendwelcher Eingabequellen beschäftigt ist, dann bleibt nicht mehr viel Rechenzeit für das eigentliche Programm über. Im Gegensatz zum 'Polling', so benennt man die Methode der dauernden Abfrage mit einem Fachausdruck, ist der Einsatz eines Interrupts in solchen Fällen wirtschaftlicher. Dann nämlich kann sich der Computer voll und

ganz seinem Programm widmen und wird erst unterbrochen, wenn eine Eingabe tatsächlich ansteht. Ein ganz konkretes Beispiel: Sobald Sie an Ihrem Atari eine Taste drücken, wird ein Interrupt ausgelöst. Das dadurch aktivierte Programm (die 'Interrupt-Routine') übernimmt den Tastencode von POKEY und trägt ihn im Speicher ein. Das erspart dem 6502 eine ganz Menge an Rechenzeit, da er sonst laufend die Tasten übeprüfen müßte, und sei es um nur zu erfahren, daß keine Taste gedrückt würde. Ein weiterer Vorteil von Interrupts: Man kann sie auch so benutzen, daß zwei Programme quasi zur selben Zeit ablaufen! Ein Beispiel zu dieser einfachsten Form von 'Multitasking' werden Sie im Laufe dieses Abschnittes noch kennenlernen.

2.3.2 Interrupts beim ATARI

Der Atari-Computer beherbergt eine ganze Reihe von Interrupts, die in drei große Gruppen aufteilbar sind:

- Ein-/Ausgabe Interrupts:
Damit zeigt z.B. die Tastatur an, daß eine Taste gedrückt wurde, weiterhin dienen diese Interrupts zum Abwickeln von I/O zu Floppy und zum Drucker.

- Timer-Interrupts:
Der Soundchip POKEY kann so programmiert werden, daß er nach einer einstellbaren Zeit einen Interrupt auslöst. Sie können sich das bildlich als eine Art 'Eieruhr' vorstellen.

- Video-Interrupts:
Damit sind zwei Interrupt-Quellen gemeint, die synchron mit der Aufbereitung des Video-Bildes ausgelöst werden. Sie erinnern sich: Das Bild auf dem Monitor bzw. Fernseher wird 50 mal in der Sekunde neu ausgegeben. Jedes dieser 50 Bilder wird von einem Elektronenstrahl in einzelnen 'Raster-Zeilen' auf den Schirm geschrieben. Der ANTIC-Baustein löst nach jedem der 50 Bilder einen Interrupt (VBI) aus, in dem u.a. die Farb-Werte aus den Schattenregistern in die Hardware-Register übertragen werden. Daneben gibt es noch den Display-List Interrupt (DLI), der am Ende einer Rasterzeile ausgelöst werden kann. Er dient zum Verändern von Hardware-Registern während des Bildaufbaues und kann dazu benutzt werden, mehr Farben, mehr Zeichensätze oder mehr beweglich Figuren darzustellen.

In den folgenden Beispielen wollen wir uns vorzüglich mit der letzteren Gruppe auseinandersetzen, da diese weitaus am häufigsten benutzt werden.

VBI: Vertical-Blank Interrupt

Mit dem VBI (Vertikal Blank Interrupt) haben wir die Möglichkeit, einen Programmteil alle 1/50-tel Sekunden ablaufen zu lassen. Jedesmal, wenn ANTIC die Daten für ein komplettes Bild ausgegeben hat, löst er einen VBI aus. Wurden keine Eingriffe vorgenommen, so läuft die InterruptRoutine des VBIs vollkommen im ROM des Betriebssystemes ab. Damit dem Benutzer eine Möglichkeit zur Verwendung des VBIs in die Hand gegeben wird, springt die Interrupt-Routine im ROM an zwei Stellen indirekt durch Vektoren im RAM. Diese beiden Vektoren sind:

VVBLKI ($222, $223): (Immediate VBI)
wird nach dem ersten Teil des VBIs durchlaufen. Er zeigt auf SYSVBV ($E45F). Dieser Sprung wird auch während zeitkritischer Operationen (wie Diskzugriff) ausgelöst.

VVBLKD ($224, $225): (Deferred VBI)
nachdem sämtliche Schattenregister kopiert wurden, erfolgt ein Sprung durch diesen Vektor. Er zeigt auf XITVBV ($E462). Während zeitkritischen Operationen wird er nicht durchlaufen.

Im Urzustand sind diese Vektoren so eingerichtet, daß sie jeweils auf den nachfolgenden Teil der ROM-Routine zeigen. Ändert man einen dieser Vektoren auf eine eigene Routine und springt nach deren Ablauf wieder ins ROM zurück, dann hat man ein zusätzliches Programmteil in den VBI 'eingebunden'. Man braucht sich bei der Benutzung dieser beiden Vektoren keine Gedanken zum 'Retten' der Register machen, dies erledigt schon die Routine im ROM für Sie. Im Beispiel dieses Abschnittes werden wir den 'Immediate'-Vektor verwenden.

Betriebssystem

Ein spezielles Problem stellt die Änderung von Interrupt-Vektoren dar. Der Grund liegt darin, daß die Vektoren zwei Bytes umfassen, und es ist nicht auszuschließen, daß ein Interrupt genau dann auftritt, nachdem man gerade ein Byte des Vektors geändert hat. Der Interrupt würde dann eine

'Mischadresse' vorfinden, die den Computer so verwirrt, daß er mit großer Wahrscheinlichkeit abstürzt. Man hilft sich entweder so, daß man die Interruptquelle anschaltet, das geht natürlich nur wenn man das geeignete Hardware-Registers kennt. Weitaus einfacher ist es, wenn man zur Veränderung von Interrupt-Vektoren eine Routine des Betriebssystems einsetzt.

> Ein Wort zur Vorsicht: Der Befehl SEI hilft nur bei sogenannten IRQs, zu denen die I/O- und Timer-Interrupts gehören, nicht aber bei den 'Non-maskable' Interrupts. Zur letzteren Gruppe zählen alle Grafik-Interrupts wie VBI oder DLI (s. später).

Man erreicht die oben genannte Routine mit einem Unterprogrammsprung zum Label SETVBV an der Adresse $E45C. Als Parameter muß ein Zeiger auf das Interrupt-Programmes übergeben werden, dessen LSB ins Y-Register und dessen MSB ins X-Register geladen wird. Im Akku wird eine Kennzahl für den gewünschten Interrupt-Vektor abgelegt: $06 bezeichnet den 'Immediate'-VBI, $07 den 'Deferred'-VBI. Ein praktisches Beispiel zu dieser Programmiertechnik finden Sie im Demo-Programm.

2.3.3 Demo-Programm zum VBI: Kurzzeituhr

Was lag näher, als mit dem periodisch wiederkehrenden VBI eine Uhr zu programmieren? Da der VBI mit ziemlicher Genauigkeit alle 1/50-tel Sekunden ausgelöst wird, geht die Interrupt-Uhr sogar relativ genau. Und - sie läuft quasi gleichzeitig mit einem anderen Programm. Wenn Sie das Listing der Kurzzeituhr mit ATMAS-II eingeben, assemblieren und (diesmal an der Adresse $600) starten, dann sehen Sie rechts oben die Uhr mit 00:00 beginnen, und gleichzeitig können Sie ATMAS-II bedienen. Sie können wieder in den Editor zurückgehen, und dort wie gewohnt arbeiten, aber Sie haben jetzt immer rechts oben die Uhr im Bild!

Nur eines sollten Sie tunlichst vermeiden: Das Programm noch mal assemblieren, wenn es bereits läuft. Denn dann schreibt ATMAS-II den Code neu, und es kann passieren, daß das noch unvollständige Programm vom noch laufenden Interrupt angesprungen wird. Daher zuvor mit RESET abschalten.

<u>Einrichten des Interrupts</u>

Die Uhr wird intern in einer Form gespeichert, die sowohl eine einfache

Möglichkeit zum Weiterzählen als auch zur Ausgabe der Uhr auf dem Schirm bietet. Es wurden dazu fünf aufeinander folgende Bytes gewählt, von denen jeder eine Ziffer (bzw. ein Zeichen) enthält.

UHR+4 : Einer-Stelle Sekunden
UHR+3 : Zehner-Stelle Sekunden
UHR+2 : interner Code für ":" minus 16
UHR+1 : Einer-Stelle Minuten
UHR : Zehner-Stelle Minuten

Diese fünf Speicherzellen werden in der Vorbereitungsphase in einer kleinen Schleife gelöscht, und der Code für den Doppelpunkt eingesetzt. Wegen der späteren Ausgabetechnik muß hier 16 vom (internen) Code subtrahiert werden. Jetzt werden X-, Y-Register sowie der Akku mit den oben aufgeführten Werten geladen und SETVBV aufgerufen. Eine 6 im Akku wählt den 'Immediate'-VBI, damit ist sichergestellt, daß die Uhr auch während Disk-Zugriffen richtig geht. Nach dem Aufruf des Betriebssystemes ist der erste Teil des Programm beendet und die Kontrolle geht per RTS an das aufrufende Programm zurück. Nach diesem kurzen Programmstück kommt ATMAS wieder zum Zuge, der ganz normal weiterarbeitet.

Die richtige Arbeit der Uhr beginnt erst anschließend. Sobald der erste Interrupt erfolgt, wird die Speicherzelle UNRUHE hochgezählt. Der CMP #50 Befehl spricht aber noch lange nicht an, so wird direkt zum Label AUSGABE verzweigt. Dort werden die Zahlen in den internen Bildschirmcode umgerechnet und dabei gleich in das Video-RAM eingeschrieben. Der Interrupt wird dann mit einem Sprung nach SYSVBV verlassen, damit übernimmt das Betriebssystem die Kontrolle. Am Rande bemerkt: Hätten wir den 'Deferred' VBI verwendet, müßte der Sprung nach XITVBV ($E462) gehen.

Ist der fünfzigste Interrupt erfolgt, so wird UNRUHE von 50 auf Null zurückgesetzt und die Einer-Stelle der Sekunden erhöht. Hat diese später ihren Maximalwert überschritten (d. h. ist sie 10), wird sie ebenfalls zurückgesetzt und die nächste Ziffer erhöht. Diese Kette setzt sich bis zur Zehner-Stelle der Minuten fort, wenn diese die 6 (für 60 Minuten) erreicht hat, wird wieder bei Null angefangen. Am Ende jedes Interrupts wird das Programm ab dem Label AUSGABE durchlaufen, so daß die Uhrzeit dauernd überschrieben wird.

Wenn nämlich das Hauptprogramm zwischen zwei Interrupts den Bildschirm löscht, wäre die Anzeige der Uhr auch verschwunden. Da sie aber 50 mal pro Sekunde neu ausgegeben wird, hat man den Eindruck, die Anzeige bliebe permanent bestehen.

```
**********************************
* Kurzzeit-UHR                V1.04
*
*   Beispiel fuer ein interrupt-
*   gesteuertes Maschinenprogramm
*   Startadresse: $0600 !
*   --> Abschalten mit RESET
*
* P.Finzel                      1986
**********************************

SAVMSC    EQU $58         Zeiger auf Video-RAM

SETVBV    EQU $E45C       Interrupt einfuegen
SYSVBV    EQU $E45F       System-Interrupt
XITVBV    EQU $E462       Ende des Interrupts
*
          ORG $600        in PAGE 6 (geschuetzt)
*
          LDA #0          Uhrzeit auf
          LDX #4          Null setzen
LOESCH    STA UHR,X
          DEX
          BPL LOESCH
          STA UNRUHE      Zaehler=0
          SEI             Maskabel Interrupts aus
          LDY #UHRVBI:L   VBI einrichten
          LDX #UHRVBI:H
          LDA #6          hier: immediate
          JSR SETVBV      VB-Interrupt
          CLI             Maskabel Interrupts an
          RTS             fertig!

**********************************
* Interrupt-Routine fuer Uhrzeit
* wird alle 1/50 sec. durchlaufen
*
* Ablauf:
* 1. Weitersetzen der Uhr
* 2. Uhrzeit in Bildschirm kopieren
**********************************

UHRVBI    CLD             Wichtig!
          LDX #0
          INC UNRUHE      50-tel Sec.
```

```
          LDA UNRUHE      volle Sekunde
          CMP #50         vorbei?
          BNE AUSGABE     keine Aenderung
          STX UNRUHE      Unruhe zurueck
          INC UHR+4       Sekunden
          LDA UHR+4
          CMP #10
          BNE AUSGABE     fertig->
          STX UHR+4
          INC UHR+3       10-Sek.-Takt
          LDA UHR+3       schon 60 Sekunden
          CMP #6          vergangen?
          BNE AUSGABE     nein, weniger ->
          STX UHR+3
          INC UHR+1       Minuten stellen
          LDA UHR+1       schon 10 Minuten
          CMP #10         vergangen?
          BNE AUSGABE     nein ->
          STX UHR+1       Minuten zurueck
          INC UHR         10-Minuten Takt
          LDA UHR         schon 6x10
          CMP #6          Minuten vorbei?
          BNE AUSGABE     nein! ->
          STX UHR         10-Minuten zurueck
*
* Ausgabe der Uhr in der rechten
* oberen Ecke des Bildschirmes
*
AUSGABE   LDA #10         Doppelpunkt-16
          STA UHR+2
          LDX #4
          LDY #39         letzte Spalte
ZIFFER    LDA UHR,X
          CLC             Umrechnung in
          ADC #16         internen Code
          ORA #$80        INVERS
          STA (SAVMSC),Y  in Video-RAM
          DEY
          DEX
          BPL ZIFFER      weiter->
          JMP SYSVBV      Interrupt beenden
*
* Variable:
*
UNRUHE    DFB 0           ;50tel-Sekunden-Takt
UHR       DFB 0,0,0,0,0   ;5 Bytes fuer Uhr
```

Kurzzeituhr (Schluß)

2.3.4 Interrupt pur: der DLI

Beim letzten Beispiel konnten alle wichtigen Aufgaben wie Register-Rettung oder Beendung des Interrupts dem Betriebssystem überlassen werden. Derartiger Komfort ist keineswegs selbstverständlich, wie Sie am nächsten Beispiel sehen werden. Gleichwohl hat man aber an diesem Programm einen guten Einblick in spezielle Techniken der Assemblerprogrammierung.

Wir wollen uns ein Beispiel zum Display-List Interrupt (DLI) genauer ansehen. DLIs können benutzt werden, um Graphik-Register während des Bildaufbaues zu ändern. Ein DLI wird ausgelöst, wenn der Elektronenstrahl an einer bestimmten Stelle am Bildschirm angekommen ist. Das Interrupt-Programm hat nun Gelegenheit, einige Hardware-Register mit neuen Werten zu besetzen. Die Graphik-Chips verwenden dann zum Zeichnen der restlichen Bildschirmzeilen die neuen Inhalte. Auf diese Weise können mehr Farben, Zeichensätze oder Players verwendet werden.

Arbeitsweise eines DLIs

Grundsätzlich unterscheidet man beim Aufbau eines Bildschirmes zwischen 'Moduszeilen' und 'Rasterzeilen'. Letztere bezeichnen den Bereich am Bildschirm, den der Elektronenstrahl bei einem Durchlauf von links nach rechts zeichnet. Eine Moduszeile dagegen kann eine oder mehrere Rasterzeilen umfassen - je nachdem, welcher Darstellungsmodus gewählt wurde. Z. B. sind die Moduszeilen eines GR. 0 Bildschirmes 8 Rasterzeilen hoch, es handelt sich um einen sog. 'Character'-(Text)-Modus. 24 Moduszeilen sind vorhanden, daraus folgt, daß 192 Rasterzeilen zum Einsatz kommen. Nicht zufällig ist das auch die vertikale Auflösung von GR.8.

In der Display-List, dem Programm des Video-Mikroprozessors ANTIC, wird für jede Moduszeile ein Code (und evtl. eine Adresse) hinterlegt. Setzt man in diesem Code das höchstwertige Bit, so löst ANTIC in dieser Zeile einen DLI aus. Aber Vorsicht: Der DLI wird erst in der letzten zur Moduszeile gehörende Rasterzeile ausgelöst und wirkt somit auf die nachfolgende Zeile! Sie sollten diesen Gesichtspunkt beim Entwurf eines Displays nie aus den Augen verlieren.

Das Demo-Programm verändert eine normale GRAPHICS 0 Display-List so, daß jede Zeile einen DLI auslöst, der wiederum die Hintergrundfarbe ändert. Auf diese Art läßt sich ein bunter Text-Bildschirm erzielen.

Bei den DLIs bekommen wir außer einem Vektor VDSLST ($200, $201) keine weitere Hilfe vom Betriebssystem. Es hilft nichts - alles muß 'per Hand' programmiert werden. Bis ein DLI seine Arbeit korrekt ausführt müssen eine Reihe von Voraussetzungen erfüllt sein:

- Pro Zeile, in der ein Display-List Interrupt stattfinden soll, muß Bit 7 in der Modus-Anweisung der Display-List gesetzt sein.
- Die Anfangsadresse der Interrupt-Routine muß in den Vektor VDSLST eingetragen werden.
- Der Interrupt muß mit Bit 7 des Hardware-Registers NMIEN ($D40E) freigegeben werden.

Für einen Text-Bildschirm GRAPHICS 0 sieht diese Display-List so aus:

```
$70  :(B)  drei mal acht
$70  :(B)  Leerzeilen zur
$70  :(B)  Zentrierung des Bildes (*)
$42  :(B)  Erste ANTIC-Anweisung für Text-Zeile
$40  :(A,  LSB) Anfangsadresse des Video-Speichers
$BC  :(A,  MSB)
$02  :(B)  weitere 23 Text-Zeilen
$02  :(B)
...     :
$02  :(B)
$41  :(B)  Sprung auf Anfang der Display-List
$00  :(A,  LSB) Anfangsadresse der Display-List
$BC  :(A,  MSB)
```

Alle mit (B) gekennzeichneten Zahlen sind Befehle für ANTIC, alle anderen mit (A) benannten Zahlen sind Zusatzinformationen zu diesen Befehlen.

Da die Wirkung eines DLIs erst am Ende einer Moduszeile auftritt, müssen wir unser Werk bereits an den Leerzeilen vor dem eigentlichen Bildschirm anfangen. Die vom Betriebssystem erzeugten Display-Lists beginnen immer mit drei Leerzeilen (zu je 8 Rasterzeilen), die nur dazu dienen, den Bildschirm zu zentrieren.

Das erste Interrupt-Bit wird im dritten Byte der Display-

List gesetzt (siehe Stern). Damit ist sichergestellt, daß der erste Farbwechsel am Beginn der ersten Textzeile erfolgt. Die folgende, sogenannte 'LMS'-Adresse darf keinesfalls verändert werden, wohl aber die nächsten 22 Modus-Anweisungen. Die letzte Zeile und der Sprungbefehl (für ANTIC, nicht verwechseln mit 6502-JMP!) müssen keinen DLI auslösen, denn der würde bereits außerhalb des sichtbaren Bildschirmes stattfinden.

Die Bits lassen sich bequem mit der indirekt-indizierten Adressierungsart und dem ORA-Befehl setzen. Wir gehen dazu so vor: Zuerst wird die Startadresse der Display-List in eine Speicherzelle der Zero-Page kopiert, das ist ja Voraussetzung für indirekten Zugriff. Nun wird das Y- Register mit einem Offset auf das gewünschte Byte geladen, und der nächste LDA (..),Y Befehl holt das Byte in den Akku. Dort wird per ORA (mit Maske $80) des Bit 7 gesetzt und das Ergebnis mit einem STA wieder an die alte Adresse zurückgeschrieben. Der ganze Vorgang wird in einer kleinen Schleife abgewickelt.

Interrupt marsch!

Als Vorsichtsmaßnahme empfiehlt es sich, anfangs den Interrupt zu sperren. Da der DLI auch zu den sog. 'nicht maskierbaren' Interrupts (NMIs) zählt, wirkt SEI nicht, man muß vielmehr das Bit 7 des Hardware-Registers NMIEN rücksetzen. Im vorliegenden Fall wäre dieser Schritt nicht unbedingt nötig, da der DLI bei GRAPHICS 0 nicht aktiv ist. Aber Vorsicht ist eben die Mutter der Porzellankiste. Nachdem der DLI sicher nicht mehr ausgelöst wird, kann man den Vektor VDSLST ohne Absturzgefahr (s. 2.3.3) auf die Interrupt-Routine richten. Danach wird das Bit 7 in NMIEN gesetzt und die DLIs sind somit scharf, die Arbeit des Programmes ist vorerst beendet.

DLI-Routinen

sind eine Wissenschaft für sich. Es ist nicht im Sinne dieses Buches, nun eine detailierte Erklärung der DLI- Problematik zu liefern, daher nur ein paar grundsätzliche Gedanken: DLIs sind ungeheuer zeitkritisch. Daher sollten sie so kurz wie nur möglich gehalten werden. Weiterhin muß man penibel aufpassen, daß man keine Register des Hauptprogrammes zerstört.

Im Beispiel (ab Label DLIPGM) wird nur der Akku verwendet. Dieser wird gleich zu Anfang mit PHA auf den Stack

geschoben und am Ende der Routine mit einem PLA wieder hergestellt. Somit ist sichergestellt, daß das Hauptprogramm nichts von der Unterbrechung mitbekommt. Die Logik der kurzen Routine sorgt nun dafür, daß pro Aufruf des Programmes ein anderer Wert in das Hardware-Register der Farbe 2 geschrieben wird. Aufgepaßt, bei DLIs dürfen Sie niemals die Schattenregister verwenden, denn diese werden erst nach dem Bildaufbau in die Hardware-Register kopiert!

Bleibt noch das kleine Problem, woher man für jede Zeile eine passende Farbe erhält. Das wurde so gelöst: Im Register VCOUNT ($D40B) kann man erfahren, welche Rasterzeile (nicht ANTIC-Moduszeile!) im Moment geschrieben wird. Genauer gesagt, steht dort nur die Rasterzeile geteilt durch zwei. Wenn dieser Wert mit vier multipliziert wird erhält man eine Zahl, die sich alle 8 Rasterzeilen in den oberen 4 Bits verändert (alles klar?!), und just wird der DLI in Abständen von 8 Rasterzeilen (gleich einer GR.0- Zeile) ausgelöst. Die unteren vier Bits werden mit AND auf Null gesetzt und erhalten mit ORA einen festen Wert für die Helligkeit.

Bevor dieser Wert seinen Weg ins Farbregister findet, braucht man noch einen Trick. Ein STA-Befehl auf das Register WSYNC ($D40A) hält den Prozessor ein Weilchen an und sorgt so dafür, daß der nachfolgende STA-Befehl (mit dem der Farbwert ins Register geschieben wird) seine Arbeit erst beginnt, wenn der Elektronenstrahl gerade rechts vom Bildschirm verschwunden ist. Würde man das nicht tun, so findet der Farbwechsel mitten in der Zeile statt, und die nimmermüde DMA von ANTIC sorgt dafür, daß es dabei furchtbar zappelig zugeht. Das DLI-Programm wird schließlich von einem RTI abgeschlossen, der die Kontrolle an das Hauptprogramm zurückgibt.

Kochrezept für DLIs

In den meisten Fällen enthält eine DLI-Routine folgende Zutaten:

1) Retten der benötigten Register (s.u.)

2) Register mit Werten laden (z.B. Farben)

3) STA-Befehl auf WSYNC

4) Register in Hardware (z.B. Farbregister) übertragen

5) Inhalte der Register vom Stack holen (s.u.)

6) RTI beendet Interrupt

In vielen Fällen braucht man mehr Register als nur den Akku, die natürlich alle auf den Stack gerettet werden müssen. Das kann man mit folgenden Befehlen erreichen

```
PHA         ;Akku retten
TXA         ;X –> A
PHA         ;X retten
TYA         ;Y –> A
PHA         ;Y retten
```

Um die Register im Schritt 4) wieder in Ordnung zu bringen, muß man sie in umgekehrter Reihenfolge vom Stack nehmen:

```
PLA         ;Inhalt Y
TAY         ;A –> Y
PLA         ;Inhalt X
TAX         ;A –> X
PLA         ;Inhalt Akku
```

Veränderungen an den Flags brauchen Sie nicht zu kümmern: Das gesamte Statusregister wird zusammen mit der Rücksprungadresse automatisch auf den Stack gerettet und durch RTI wieder restauriert.

Mit diesem Handwerkszeug können Sie die meisten DLI Aufgaben lösen. Vielleicht noch ein paar Tips zum Schluß: Im Beispiel hatten wir in jeder Zeile den gleichen DLI ausgelöst. Man kann natürlich bei anderen Problemstellungen nur einen einzigen DLI zulassen, oder mehrere verschiedene DLI-Routinen benutzen. Im letzteren Fall braucht man nur zwischen Schritt 4 und 5 die Adresse der nächsten Routine in VDSLST eintragen. Dann aber nicht vergessen, beim letzten DLI wieder die Adresse des ersten einzutragen!

```
**********************************
*      DISPLAY-LIST-INTERRUPTS V1.12
*
*   Farbiges Demo einer Interrupt-
*          Service-Routine
*
* P.FINZEL                      19B6
**********************************

COLPF2     EQU $D018       Hardware-Reg. fuer Farbe
WSYNC      EQU $D40A       Warten auf Zeilenende
```

```
VCOUNT    EQU $D40B       Rasterzeile durch 2
NMIEN     EQU $D40E       NMI-Freigabe

VDSLST    EQU $0200       Vektor fuer DLIs
SDLSTL    EQU $0230       Zeiger auf D-List

HILFZP    EQU $80         Hilfsregister in Zero-P.

          ORG $A800

VORBERT   LDA SDLSTL      Display-List-Zeiger in
          LDY SDLSTL+1    Zero-Page kopieren
          STA HILFZP      fuer indirekte
          STY HILFZP+1    Adr.-Art
          LDY #2          DLI-Bit in
          LDA (HILFZP),Y  dritter Leerzeile
          ORA #$80        setzen
          STA (HILFZP),Y
          INY             DLI-Bit in der
          LDA (HILFZP),Y  LMS-Anweisung
          ORA #$80        der D-List setzen
          STA (HILFZP),Y
          INY             LMS-Adresse
          INY             ueberspringen
INTSET    INY             Schleife fuer
          LDA (HILFZP),Y  DLI-Bit in restl.
          ORA #$80        Display-List
          STA (HILFZP),Y
          CPY #27         schon alle Zeilen?
          BNE INTSET      nein-->
*
          LDA #$40        zur Vorsicht DLI
          STA NMIEN       erst abschalten
          LDA #DLIPGM:L   den Vektor auf
          STA VDSLST      die Int.-Routine
          LDA #DLIPGM:H   setzen
          STA VDSLST+1
          LDA #$C0        und den DLI
          STA NMIEN       wieder freigeben
          RTS
*
* Interrupt-Routine zur Bedienung
* des DLIs
*
DLIPGM    PHA             Akku retten
          LDA VCOUNT      Rasterzeile
          ASL             ;mit vier
          ASL             ;multiplizieren
          AND #$F0        Farbe maskieren
          ORA #$04        Helligkeit
          STA WSYNC       HBlank abwarten
          STA COLPF2      Farbe setzen
          PLA             Akku besorgen
          RTI             Schluss!
```

DLI-Programm (Schluß)

2.4 Vom Umgang mit Objektprogrammen

Unsere bisherige Vorgehensweise war es, den Objektcode stets durch direkte Assemblierung in den Speicher zu schreiben und diesen schließlich mit dem ATMAS-Monitor zu starten. Das ist zwar zur Entwicklung und zum Testen eines Programmes sehr nützlich und praktisch, aber wenn das Programm einmal fertiggestellt ist, möchte man es nicht in einer Form aufbewahren, in der es jedesmal zuvor assembliert werden muß.

2.4.1 SAVE-Funktion des ATMAS-Monitors

Kein Problem, denn dazu bietet der ATMAS-Monitor seine SAVE-Funktion an. Sie erlaubt, einen zusammenhängenden Speicherbereich auf Diskette abzulegen. Das Format der gespeicherten Disk-Datei ist so gewählt, daß das File mit der DOS-Funktion 'L' (für Binary-Load) wieder geladen werden kann. Natürlich kann man auch die LOAD-Option des ATMAS-Monitors benutzen.

Für die SAVE-Funktion müssen Sie Anfangs- und Endadresse des Ojektprogrammes kennen. Die Endadresse zeigt Ihnen ATMAS-II nach vollendeter Assemblierung an, und als Anfangsadresse verwenden Sie im Regelfall die Adresse bei ORG.

Die SAVE-Funktion bietet dazu noch die Möglichkeit des adressversetzten Abspeicherns durch die INTO-Adresse an. Man macht von dieser Option Gebrauch, wenn man ein Maschinenprogramm an eine Stelle assemblieren will, die während der Entwicklungszeit z. B. durch ATMAS-II oder durch das DOS belegt ist. Wenn Sie bei 'INTO' eine Adresse angeben, wird das Programm so aufgezeichnet, daß es beim nächsten Ladevorgang ab dieser Adresse geladen wird. Wenn Sie das Programm zuvor mit diesem Wert als 'logische' Adresse beim ORG-Befehl assembliert haben, dann haben Sie jetzt ein lauffähiges Maschinenprogramm auf Diskette. Diesen Vorgang finden Sie auch im ATMAS-Handbuch (/5/ auf Seite 20 und 33f) ausführlich beschrieben.

Segmentierte Files

Es kann auch der Fall auftreten, daß ein Objektprogramm nicht aus einem durchgehenden Block, sondern aus mehreren verstreut im Speicher liegenden Einzelbereichen besteht.

Nehmen wir als Beispiel an, daß ein Teil ab Adresse $A800 (bis $AB00) läge, ein zweiter in der Page 6. Wie im ATMAS- Handbuch beschrieben, speichern wir den ersten Teil mit den Eingaben:

Ihre Eingabe	Bildschirm
S	SAVE
A800	FROM:A800
AB00	TO :AB00
<RETURN>	INTO:
D:TEST.OBJ	FILENAME:D:TEST.OBJ

Anschließend wird der zweite Teil (aus Page 6) durch einen SAVE-Befehl angehängt. Man verwendet hierbei die APPEND- Option, die mit einem ">"-Zeichen am Ende des Filenames gekennzeichnet wird:

Ihre Eingabe	Bildschirm
S	SAVE
0600	FROM:0600
06FF	TO :06FF
<RETURN>	INTO:
D:TEST.OBJ>	FILENAME:D:TEST.OBJ>

Durch diese kleine Ergänzung wird das zweite File an das erste 'angeklebt'. Das entstandene File nennt man 'segmentiertes' oder 'Compound'-File. Beim Ladevorgang wird die Segmentierung automatisch beachtet. Ein File kann aus beliebig vielen Segmenten bestehen.

Diese Fähigkeit bietet mehr Möglichkeiten als man zunächst vermuten könnte. Man kann damit z. B. mehrere kleinere Objektprogramme zu einem großen zusammenfügen und mit einigem Geschick Zeichensätze, Hi-Res Bilder oder PM-Daten von anderen Programmen (z.B. von Zeichensatz-Editoren oder Graphik-Programmen) einbinden. Wie Sie in den nächsten beiden Abschnitten noch sehen werden, kann man es auch als Ablaufsteuerung beim Ladevorgang benutzen.

2.4.2 Auto-Start mit RUN-Adresse

Ein nach dem bisher beschriebenen Muster gespeichertes Maschinenprogramm wird jedoch nach dem Ladevorgang noch nicht automatisch gestartet. Nach dem Laden würde wieder DOS aktiviert und Sie müßten das Programm mit DOS-Funktion 'M' (Run at Address) unter

Eingabe seiner Startadresse laufen lassen. Das wäre recht mühselig, außerdem müßte man dabei die Startadressen (die ja nicht mit den Anfangsadressen übereinstimmen müssen) aller Programme im Kopf behalten.

Das DOS 2.5 (und auch DOS 2.0) bietet aber auch die Möglichkeit, ein Binärfile automatisch nach dem Laden zu starten. Das funktioniert so: Nachdem das Programm komplett in den Speicher geladen wurde, springt DOS durch den Vektor RUNAD ($2E0, 2E1). Hat man zuvor diese beiden Speicherzellen durch ein nur aus zwei Bytes bestehendes Segment überschrieben und auf die Startadresse gerichtet, so wird das Programm unverzüglich nach dem Ladevorgang ausgeführt.

Eine RUN-Adresse kann man mit der APPEND-Option des SAVE-Befehles erzeugen. Zur Vorbereitung fügt man am Schluß des Quelltextes noch zwei Zeilen an:

```
ORG $2E0
DFW START
```

START sei hier der Label, der den Einsprung des Maschinenprogrammes angibt. Beim Assemblieren wird nun gleich die RUN-Adresse in den Vektor eingetragen, wir ersparen uns somit die Arbeit, dies per Hand im Monitor vornehmen zu müssen. Nehmen wir nun an, das Programm wäre schon unter dem Namen TEST.OBJ gespeichert. Die RUN-Adresse wird dann so angefügt:

Ihre Eingabe	Bildschirm
S	SAVE
02E0	FROM:02E0
02E1	TO :02E1
<RETURN>	INTO:
D:TEST.OBJ>	FILENAME:D:TEST.OBJ>

Wichtig ist dabei die Eingabe des ">"-Zeichens nach dem Filenamen.

2.4.3 Vorspann mit INIT-Adresse

Sie wissen bereits, daß ein Binär-File aus mehreren Segmenten bestehen kann. Mit der INIT-Adresse kann man erreichen, daß schon während des Ladevorganges gewisse Programmteile ausgeführt werden. Man kann somit Programme schreiben, die z.B. kurz nach dem Beginn des Ladevorganges

schon ein Titelbild zeigen. Immer nachdem ein Segment geladen wurde, verzweigt DOS durch den Vektor INITAD ($2E2, $2E3). Hat man ein kleines Segment eingebaut, das diesen Vektor überschreibt, so kann man ein Programm vor dem Laden des nächsten Segmentes starten. Natürlich muß man dafür Sorge tragen, daß das angesprungene Programm schon zuvor im Rahmen eines Segmentes in den Speicher geladen wurde. Der Ladevorgang wird forgesetzt, sobald das Programm mit einem 'RTS' zum DOS zurückkehrt.

Eine INIT-Adresse kann man auf dieselbe Weise wie eine RUN-Adresse ans File anhängen, nur tritt an die Stelle von $2E0 die Adresse $2E2. Die INIT-Adresse ist somit ideal zur Programmierung von Vorspann-Routinen geeignet, die längere Ladezeiten überbrücken. Übrigens arbeitet der Vorspann von ATMAS-II auf die gleiche Weise.

Ein Trick am Rande: Bindet man durch eine INIT-Routine ein Programm in den 'Immediate'-VBI ein, dann läuft die Vorspann-Routine während des Ladevorganges weiter, und man kann z.B. Farben ändern oder die PM-Graphik zur Animation einsetzen. Es ist nicht ganz einfach, aber es ist sicher eine reizvolle Aufgabe für einen angehenden Assemblerprogrammierer. Die Mühe lohnt sich bestimmt.

2.4.4 Automatisches Booten

Wenn Sie ein selbststartendes Maschinenprogramm nach den gerade beschriebenen Regeln hergestellt haben, dann können Sie es auch so speichern, daß es gleich nach dem Einschalten des Computers geladen und gestartet wird. Alles was Sie brauchen ist eine formatierte Diskette mit DOS 2.5 oder 2.0. Kopieren Sie darauf ihr Binärfile (z.B. mit DOS- 'O') und nennen Sie es mit DOS-'E' (Rename) in

AUTORUN.SYS

um. Wenn Sie wollen, können Sie auch DUP.SYS von der Disk löschen, es wird nicht benötigt.

Nun schalten Sie den Computer aus, drücken <OPTION> und schalten wieder ein. Wenn alles geklappt hat, dann wird Ihr Programm vollkommen ohne zutun geladen und gestartet. Auf diese Art können Sie Ihren Programmen ein 'profi'-mäßiges Aussehen verleihen!

2.5 Kopplung von BASIC mit Maschinenprogrammen

Bestimmt ist es nicht der schlechteste Anfang, vor der Programmierung von reinen Maschinenprogrammen, erst kleinere Assemblerroutinen zur Unterstützung von BASIC-Programmen zu schreiben. Tatsächlich haben so die meisten der Leute angefangen, die heute große Unterhaltungs- und Anwendungsprogramme in Assembler schreiben. Bei der Verbindung von BASIC mit Assemblerprogrammen gibt es drei wichtige Punkte zu berücksichtigen:

1.) Speicherplatz: welche Adressen sind für das Maschinenprogramm reserviert?
2.) Verknüpfung: Wie funktioniert die Verständigung zwischen BASIC und Maschinenprogramm?
3.) Integration: wie kommen BASIC und Maschinenprogramm von Diskette möglichst automatisch in den Speicher?

Ein allgemein gültiges Kochrezept läßt sich für diese drei Fragen nicht geben, da die Verknüpfung BASIC-Assembler je nach Art und Länge des Maschinenprogrammes anders ausfallen dürfte.

2.5.1 Speicherplatz

Wohin mit den Maschinenprogrammen? Für kurze Programme bietet sich die Page 6 an, ein von DOS und Betriebssystem reservierter Bereich von $6000 bis $6FF (256 Bytes). Dieser Bereich ist sowohl in BASIC als auch unter ATMAS-II frei verfügbar.

Für längere Maschinenprogramme wird es etwas schwieriger. Der bei ATMAS-II reservierte Bereich ab $A800 ist in BASIC nicht verfügbar, denn dort liegt gerade das BASIC-ROM. Man muß die Maschinenprogramme für einen tiefer liegenden Speicherbereich assemblieren, das läßt sich leicht mit der 'logischen' Adresse beim ORG-Befehl erreichen. Schreibt man

```
ORG $7000,$A800
```

so wird das Programm zwar ab Adresse $A800 im Speicher abgelegt, aber so assembliert, daß es an der Adresse $7000 lauffähig ist (s. auch /5/ Seite 20)

Damit hat man die Möglichkeit, die Programm praktisch frei im Speicher zu positionieren. Man muß natürlich aufpassen, daß man keine Überschneidungen mit dem von BASIC oder dem Videospeicher beanspruchten Bereichen bekommt.

Die einfachste und auch unsicherste Methode zur Kopplung von BASIC mit Assembler besteht darin, daß man das Maschinenprogramm einfach mitten ins RAM schreibt (z.B. ab $7000) und dann hofft, daß weder das BASIC-Programm zu lange wird, noch der Graphikspeicher das Ende des Maschinenprogrammes überschreibt. Eine solche Vorgehensweise ist allerdings nur für schnelles Ausprobieren zu empfehlen.

Etwas schwieriger wird es schon, wenn man sich an die Spielregeln des Betriebssystemes hält, und vor dem Benutzen eines Speicherbereiches die Zeiger auf den freien Speicher verschiebt. So kann man z.B. den Zeiger auf die untere Speichergrenze MEMLO ($2E7, $2E8), der normalerweise auf das Ende des DOS zeigt, nach oben verschieben. Diese Aufgabe muß von einem AUTORUN.SYS-File übernommen werden, das ausgeführt wird, bevor BASIC zum Zuge kommt.

Eine andere beliebte Möglichkeit ist die Verschiebung von RAMTOP ($6A) nach unten, so daß Platz oberhalb des Bildschirmspeichers reserviert wird. Das funktioniert auch in BASIC: Mit

```
POKE 106,144:GRAPHICS 0
```

werden 16 Pages (4KByte, RAMTOP enthält 160 mit BASIC ein) ab der Adresse 144*256=36864 ($9000) reserviert. Es empfiehlt sich jedoch, die ersten 256 Bytes nicht zu verwenden, da diese durch einen Fehler im Bertriebssytem überschrieben werden können (s. auch /1/ S.101).

2.5.2 Der USR-Befehl

Genauso wie ein Maschinenprogramm im ATMAS-Monitor mit dem GOTO-Befehl gestartet wird, kann man das in BASIC mit dem USR-Befehl erledigen. Zusätzlich erlaubt USR die Übergabe von Zahlen ('Parametern') von BASIC zum Maschinenprogramm und auch umgekehrt. Damit kann man das Maschinenprogramm mit Daten versorgen, bzw. Ergebnisse zurückmelden lassen.

Wir wollen uns die Funktionen an einem einfachen Beispiel erarbeiten.

Dazu soll eine Funktion definiert werden, die zwei aufeinanderfolgende Bytes als ein Wort aus dem Speicher liest, mehr oder minder eine bessere PEEK-Funktion. Da das Programm sehr kurz ist, kann es in Page 6 abgelegt werden. Der Aufruf erfolgt mit:

```
X=USR(1536,ADR)
```

Folgende Aktionen werden ausgelöst: Der (in diesem Fall) einzige Parameter wird als 2-Byte Integer (Wort) auf dem Stack abgelegt, dann wird noch ein Byte auf den Stack geschoben, das die Anzahl der Parameter vermerkt, im vorliegenden Fall eine Eins. Jetzt wird das Programm an der Adresse 1536 gestartet und verrichtet seine Arbeit. Es endet mit einem RTS und gibt somit die Kontrolle an BASIC zurück. Hat das Maschinenprogramm sein Resulat in die Speicherzellen FR0 ($D4, LSB) und FR0+1 ($D5, MSB) eingetragen, so findet sich dieser Wert in der Variablen X wieder.

In einem Maschinenprogramm, das mit USR gestartet werden soll, ist daher folgendes zu tun:

1.) Den Wert für die Anzahl der Parameter vom Stack nehmen. Auch wenn keine Parameter übergeben werden, ist dieses Byte vorhanden! Daher grundsätzlich erster Befehl: PLA
2.) Dann alle Parameter (falls vorhanden) mit PLA-Befehlen vom Stack nehmen. Das erste PLA liefert immer das MSB, der zweite PLA das LSB. Die Parameter erscheinen genau in der im USR Befehl angegebenen Reihenfolge.
3.) Falls nötig wird das Ergebnis in FRO ($D4, LSB) und FRO+1 (MSB) eingetragen. Diese Integerzahl wird später von BASIC in die Variable vor dem '='-Zeichen von USR geschrieben. Falls Sie nichts zurückgeben wollen, dann einfach nichts nach FRO schreiben und auch die Ergebnis-Variable nicht beachten.
4.) RTS ausführen. Dieser Schritt geht natürlich nur gut, wenn die Schritte 1 und 2 ordnungsgemäß erledigt wurden.

Man sieht: ein wenig Umsicht ist bei USR-Routinen nicht fehl am Platze. Achten Sie daher sorgsam auf die PLA-Befehle und auf die korrekte Anzahl der Parameter im USR- Befehl. Ein konkretes Beispiel zur Programmierung

von USR-Routinen finden Sie im Anschluß an diesen Abschnitt ('Doppel-Peek'). Außerdem finden Sie dort das Programm in Form eines Basic-Loaders. Dieses Listing können Sie mit BASIC eintippen und so das Zusammenspiel BASIC Maschinenprogramm ausprobieren.

2.5.3 Integration

Damit wäre bereits das nächste Thema angeschnitten: Wie schafft man es, BASIC und Maschinenprogramm zu einem gemeinsamen Programm zu vereinen? Es gibt mehrere Möglichkeiten:

1.) getrenntes Laden von BASIC und Maschinenprogramm von Hand
2.) Einfügen des Maschinenprogrammes als DATA-Zeilen oder Strings

Wieder ist die Möglichkeit 1 die einfachste, wenn gleich sie eine Menge an Eingaben erfordert. Man geht so vor: DOS-Diskette mit BASIC booten, 'DOS' eingeben und das Maschinenprogramm mit 'L'-Befehl laden. Dann mit 'B' ins BASIC zurück und mit 'LOAD...' das BASIC-Programm laden. Das Objektprogramm muß zuvor mit dem ATMAS-II Monitor abgespeichert worden sein, wie es im Abschnitt 2.4. erklärt wurde (ohne RUN und INIT-Adressen).

Diese Vorgehensweise eignet sich natürlich nur während der Testphase oder einfach zum Ausprobieren. Will man dagegen Programme machen, die professionell aussehen, dann muß alles ohne viel Eingabe laufen. Hier bietet sich an, das Maschinenprogramm im BASIC-Teil aufzubewahren und es erst beim Ablauf von BASIC per POKE-Befehle in den Speicher zu transferieren. Zum Aufbewahren der Programmcodes eignen sich DATA-Zeilen oder Strings. Natürlich wäre es sehr mühsam, alle Codes per Hand aus dem Assemblerlisting in DATA-Zeilen zu übertragen. Besser benutzt man dazu ein kleines Programm, das aus einem Objektprogramm auf Diskette die DATA-Zeilen selbständig generiert. Für alle, die mein erstes Buch 'Die Hexenküche' noch nicht haben, ist das dort enthaltene Programm DATGEN im Anhang nochmal abgedruckt.

Dieses erzeugt nicht nur die DATA-Zeilen, sondern auch gleich eine FOR-NEXT-Schleife mit den passenden Adressen.

Verwendung: Maschinenprogramm mit ATMAS-Monitor auf Diskette speichern. Dann erneut mit BASIC booten und DATGEN laden. Nun den Filenamen des Maschinenprogrammes eingeben, einen Filenamen für das erzeugte BASIC-Programm und dessen Anfangszeilennummer eintippen. Alles weitere läuft dann von selbst. Das erzeugte BASIC-Programm kann man problemlos mit ENTER"D:..." zu jedem anderen Programm dazubinden. Als Aufruf genügt ein GOSUB auf die oben eingegebene Zeilenummer - und - ein USR-Befehl zum Aufruf des Maschinenprogrammes.

```
*********************************
* Doppel-Peek (Deek) Befehl   V1.00
*
* Aufruf: D=USR(1536,<Adresse>)
*
*********************************
*
FRO       EQU $D4         Rueckgabe USR-Ergebnis
HILF      EQU $CB         Hilfsregister in Zero-Page
*
          ORG $0600       in Page 6
*
          PLA             Anzahl der Parameter
          PLA             Adresse MSB vom Stack
          STA HILF+1      in Zero-Page
          PLA             LSB der Adresse
          STA HILF        in Zero-Page
          LDY #0          Offset gleich null
          LDA (HILF),Y    LSB des Datums
          STA FRO         in Ergebnis-Register
          INY             Offset auf MSB
          LDA (HILF),Y    MSB laden
          STA FRO+1       ins Erg.-Register
          RTS             fertig
```

Assembler-Listing: 'Doppel-Peek'

```
100 REM LISTING 2: BASIC-Loader fuer USR-DEMO
110 GOSUB 1000
120 DEEK=1536
130 ? "Der Screen beginnt bei ";USR(DEEK,88)
190 END
1000 REM * BASIC-LOADER fuer DEEK
1010 S=0:RESTORE 1100
1020 FOR A=1536 TO 1554:READ D:POKE A,D:S=S+D:NEXT A
1030 IF S<>2892 THEN ? "DATEN-FEHLER!":STOP
1090 RETURN
1100 DATA 104,104,133,204,104,133,203,160,0,177,203,133,212,200,177
1110 DATA 203,133,213,96
```

BASIC-Loader zu Doppel-Peek (erzeugt mit DATGEN)

2.6 MAKROS

Beim Programmieren trifft man häufig auf Probleme, die man bei einem früheren Programm schon gelöst hatte. Nun gibt es zwei Alternativen: Entweder man erinnert sich, wie man das denn damals gemacht hat, oder, zweite Möglichkeit, man hat sich schon beim ersten Versuch in weiser Voraussicht ein Makro angelegt, das man jetzt sofort wieder einsetzen kann. Klar, daß die zweite Lösung eine Menge Zeit spart, die ja für manche Leute Geld sein soll...

Damit wären wir bei einer Funktion von ATMAS-II, die wir bisher noch überhaupt nicht beachtet haben: die Makrofähigkeit. Makros kann man als eine Art Schablone für eine Befehlsfolge verstehen, daher erzeugt die Definition eines Makros noch keinen Objektcode. Sehen wir uns dazu das Beispiel einer einfachen Makro-Definition an, die zum 'Retten' der Prozessorregister in einem DLI dienen könnte:

```
PHR     MACRO
        PHA
        TXA
        PHA
        TYA
        PHA
        MEND
```

Wenn Sie diese Befehlsfolge von ATMAS-II assemblieren lassen, dann werden Sie feststellen, daß kein einziges Byte an Objektcode erzeugt wurde. Erst wenn man das Makro benutzt, d.h. dessen Namen wie einen Assemblerbefehl in den Quelltext einfügt, dann werden die Befehle der 'Schablone' eingesetzt. Hier ein Beispiel, das z. B. aus einer DLI- Routine stammen könnte:

```
Quelltext:          wird zu:
...                 ...
                    PHA
                    TXA
PHR                 PHA
                    TYA
                    PHA
LDA #$C0            LDA #$C0
LDX #$A4            LDX #$A4
STA WSYNC           STA WSYNC
```

...

Hat man das Makro erst einmal definiert, so braucht man nur PHR (soll für 'PusH Registers' stehen) zu schreiben, wenn man im Programm alle Register auf den Stack schieben will. Der Assembler ersetzt den Makroaufruf durch die definierten Maschinenbefehle, man bezeichnet diesen Vorgang als 'Makro-Expansion'. Wenn man es so will, dann hat man mit dem Makro einen zusätzlichen 'Befehl' geschaffen, der aus mehreren (Elementar-) Befehlen zusammengesetzt ist.

Formale Parameter

Zugegeben, wenn das alles wäre, dann würden Makros zu nicht viel nütze sein. Wesentlich flexibler werden Makros durch eine Einrichtung namens 'formale Parameter'. Am Anfang einer Definition kann man einen Satz symbolischer Namen angeben, die in den folgenden Befehlen des Makros wie Labels verwendet werden dürfen. Es handelt sich dabei nicht um wirkliche Labels, sondern nur um Platzhalter für Werte, die beim Aufruf des Makros angegeben werden. Wir schauen uns diesen Sachverhalt am besten an einem Beispiel an:

```
SETCOLOR MACRO REGISTER,FARBE,HELL
         LDA #FARBE
         ASL
         ASL
         ASL
         ASL
         ORA #HELL
         LDX #REGISTER
         STA $2C4,X
         MEND
```

Nehmen wir an, wir brauchen einen Befehl zum Verändern der Farbregister. In BASIC geschieht das durch den Befehl 'SETCOLOR', daher bietet sich dieser Name auch für das Makro an. In BASIC geben wir dem Befehl drei Parameter auf den Weg, die Farbe, Helligkeit und Register auswählen. Beim Makro definieren wir drei formale Parameter, im Beispiel REGISTER, FARBE und HELL (für Helligkeit) genannt.

Die Funktion des Makros dürfte Ihnen nach der bisherigen Lektüre des Buches sofort klar sein: Der Akku wird mit 16 multipliziert und damit die

Farbinformation in die oberen vier Bits verschoben. Die unteren vier Bits werden durch Oder-Verknüpfung mit HELL hergestellt und das Ergebnis via X-Indizierung in die Schattenregister der Farbe ab $2C4 eingetragen.

> Am Rande bemerkt: An dieser Stelle könnten Sie einen Stilbruch vermuten, da weiter vorne gepredigt wurde, keine Zahlen im Quelltext zu verwenden, sondern dafür symbolische Namen per EQU zu definieren. Im vorliegenden Fall wurde absichtlich darauf verzichtet, um das Makro sehr kompakt zu halten. Will man nämlich das Makro später auch in anderen Programmen einsetzen, so dürfte man nie vergessen, die EQU-Definition (die ja außerhalb des Makros stehen würde) auch mit zu übernehmen. Schreibt man die EQU-Definition der Adresse ins Makro hinein, so besteht die Gefahr einer Doppeldefinition, falls der Name außerhalb des Makros schon vergeben wäre!

Sehen wir uns an, was beim Aufruf des Makros passiert:

```
Quelltext:                wird zu:
GRUEN EQU $0B
...                       LDA #GRUEN
                          ASL
                          ASL
                          ASL
                          ASL
SETCOLOR 2,GRUEN,$0A      ORA #$0A
                          LDX #2
                          STA $2C4,X
LDA #100                  LDA #100
...                       ...
```

Alle formalen Parameter der Definition wurden beim Aufruf durch die 'tatsächlichen' Parameter ersetzt, und diese Befehlsfolge wird assembliert. Zu beachten ist natürlich, daß die Adressierungsart durch das Makro vorgegeben wird. Das bedeutet, daß im Beispiel nur Konstanten übergeben werden dürfen. Wollte man Adressen übergeben, so müßte das Makro modifiziert werden und statt Immediate- die absolute Adressierung gewählt werden (d.h. LDA FARBE ... ORA HELL...).

Lokale Labels

Sobald Sie ein Label innerhalb eines Makros verwenden müssen, meldet der Assembler einen Fehler, wenn das Makro mehr als einmal verwendet wurde. Mit gutem Grund, denn durch das Einsetzen des Makros taucht nun das gleiche Label an mehreren Stellen im Programm auf. Da man Labels im Makro für Verzweigungen oder Schleifen dringend braucht, bietet ATMAS-II hier eine spezielle Sorte von Labels an, die nur innerhalb des Makros gültig sind.

Ein lokales Label endet im Unterschied zu einem normalen Label mit dem 'AT'-Symbol, das durch die Tastenkombination <Shift>-8 erzeugt wird (auch Happy-'a' genannt). Dieses Symbol wird beim Assemblieren durch eine vierstellige Zahl ersetzt, die vor jedem Makroaufruf erhöht wird. Auf diese Art erzeugen zwei Aufrufe des gleichen Makros unterschiedliche Labels (s. auch /5/ S. 27).

Viel Speicherplatz

Wir haben uns ein Makro als eine Art Schablone aus Assemblerbefehlen und formalen Parametern vorgestellt, die beim Aufruf in das Programm eingesetzt wird. Daraus folgt aber auch, daß beim mehrmaligen Gebrauch das Makro immer neu eingesetzt und auch dementsprechend Speicherplatz verbraucht wird.

Hier zeigt sich der deutliche Unterschied zu Unterprogrammen: Während ein Makro bei jedem Aufruf ins Programm eingesetzt wird, ist ein Unterprogramm nur einmal vorhanden. Zum Aufruf wird dann nur ein drei Bytes langer Sprung (JSR) eingesetzt.

Auf der anderen Seite kann ein Programm durch den Einsatz von Makros übersichtlicher gestaltet werden. Es ist sogar möglich, ganze Programmiersprachen als Makros zu implementieren (sog. Makro-Sprachen).

LITERATURVERZEICHNIS

/1/ Finzel, Die Hexenküche. 1984, Verlag P. Finzel Productions Fürth/Bay.

/2/ Reschke/Wiethoff, Das Atari Profibuch, 1985, Sybex-Verlag Düsseldorf

/3/ Crawford u.a., De Re Atari, 1981, 0.0.

/4/ MOS-Microcomputers Programmierhandbuch, MOS Technology, o.J., o.B.

/5/ P.Finzel, ATMAS-II Handbuch, 1985, P. Finzel Productions, Fürth/Bay.

ANHANG A: Befehlssatz des 6502 (Codes sind hexadezimal)

Befehl	Relativ	Implied	Immediate	Zero-Page	Zero-Page,X	Zero-Page,Y	Absolut	Absolut,X	Absolut,Y	(Indirekt,X)	(Indirekt),Y	Indirekt
ADC			69	65	75		6D	7D	79	61	71	
AND			29	25	35		2D	3D	39	21	31	
ASL		0A		06	16		0E	1E				
BCC	90											
BCS	B0											
BEQ	F0											
BIT				24			2C					
BMI	30											
BNE	D0											
BPL	10											
BRK		00										
BVC	50											
BVS	70											
CLC		18										
CLD		D8										
CLI		58										
CLV		B8										
CMP			C9	C5	D5		CD	DD	D9	C1	D1	
CPX			E0	E4			EC					
CPY			C0	C4			CC					
DEC				C6	D6		CE	DE				
DEX		CA										
DEY		88										
EOR			49	45	55		4D	5D	59	41	51	
INC				E6	F6		EE	FE				
INX		E8										
INY		C8										
JMP							4C					6C
JSR							20					
LDA			A9	A5	B5		AD	BD	B9	A1	B1	
LDX			A2	A6		86	AE		BE			
LDY			AO	A4	B4		AC	BC				
LSR		4A		46	56		4E	5E				
NOP		EA										
ORA			09	05	15		0D	1D	19	01	11	
PHA		48										
PHP		08										
PLA		68										
PLP		28										
ROL		2A		26	36		2E	3E				
ROR		6A		66	76		6E	7E				
RTI		40										
RTS		60										
SBC			E9	E5	F5		ED	FD	F9	E1	F1	
SEC		38										
SED		F8										
SEI		78										
STA				85	95		8D	9D	99	81	91	
STX				86		96	8E					
STY				84	94		8C					
TAX		AA										
TAY		A8										
TYA		98										
TSX		BA										
TXA		8A										
TXS		9A										

ANHANG B-1: Ausführungszeiten in Maschinenzyklen

Befehl	Relativ	Implied	Immediate	Zero-Page	Zero-Page,X	Zero-Page,Y	Absolut	Absolut,X	Absolut,Y	(Indirekt,X)	(Indirekt),Y	Indirekt
ADC			2	3	4		4	4+	4+	6	5+	
AND			2	3	4		4	4+	4+	6	5+	
ASL		2		5	6		6	7				
BCC	2#											
BCS	2#											
BEO	2#											
BIT				3			4					
BMI	2#											
BNE	2#											
BPL	2#											
BRK		7										
BVC	2#											
BVS	2#											
CLC		2										
CLD		2										
CLI		2										
CLV		2										
CMP			2	3	4		4	4+	4+	6	5+	
CPX			2	3			4					
CPY			2	3			4					
DEC				5	6		6	7				
DEX		2										
DEY		2										
EOR			2	3	4		4	4+	4+	6	5+	
INC				5	6		6	7				
INX		2										
INY		2										
JMP							3					5
JSR							6					
LDA			2	3	4		4	4+	4+	6	5+	
LDX			2	3		4	4		4+			
LDY			2	3	4		4	4+				
LSR		2		5	6		6	7				
NOP		2										
ORA			2	3	4		4	4+	4+	6	5+	
PHA		3										
PHP		3										
PLA		4										
PLP		4										
ROL		2		5	6		6	7				
ROR		2		4	6		6	7				
RTI		6										
RTS		6										
SBC			2	3	4		4	4+	4+	6	5+	
SEC		2										
SED		2										
SEI		2										
STA				3	4		4	5	5	6	6	
STX				3		4	4					
STY				3	4		4					
TAX		2										
TAY		2										
TYA		2										
TSX		2										
TXA		2										
TXS		2										

\+ Addiere einen Zyklus, wenn Pagegrenze überschritten wird.
\# Addiere einen Zyklus, wenn Verzweigung stattfindet.
Addiere noch einen Zyklus, wenn Pagegrenze überschritten wird.

ANHANG B-2: Ermittlung der Rechenzeit.

Wenn Sie wissen möchten, wie lange die Ausführung eines Programmes oder eines Programmteiles dauert, so können Sie das anhand der Tabelle im Anhang B-1 ermitteln. Zu jedem Befehl müssen Sie unter Berücksichtigung der Adressierungsart die Ausführungszeit in Zyklen heraussuchen. Zählen Sie dann alle Zyklen zusammen, wobei natürlich beachtet werden muß, ob ein Programmteil übersprungen oder mehrmals ausgeführt wird.

Der 6502 des Atari-Computers wird mit 1, 79 MHz getaktet, das bedeutet, daß er 1790000 Zyklen in einer Sekunde ausführt. Beispiel: Braucht ein Maschinenprogramm 20000 Zyklen, so dauert die Ausführung:

$$\frac{20000}{17900000} = 0{,}011 \text{ sec.}$$

oder ca. 11 Millisekunden. Allerdings ist dieser Wert beim Atari rein theoretischer Natur, da die maximale Rechengeschwindigkeit durch einige Effekte wie Refresh, DMA und auch dem VBI enorm gebremst wird (ausführliche Erklärung: /1/ S. 48ff). Bei eingeschaltetem Bildschirm (GRAPHICS 0) werden nur noch ca. 1160000 Zyklen pro Sekunde ausgeführt. Dieser Wert ist stark vom Darstellungsmodus des Bildschirmes abhängig, so daß Zeitberechnungen beim Atari nur bedingte Gültigkeit haben.

ANHANG C: Wichtige Adressen

Nachfolgend eine Tabelle der wichtigsten in diesem Buch benutzten Speicherzellen zusammen mit dem gewöhnlich verwendeten Labelnamen. Sehr hilfreich beim Programmieren in Assembler ist eine vollständige Memory-Map, wie sie z. B. in /2/ ab S.9ff zu finden ist.

Page 0 und Page 2

RTCLOK	$012-$014	Uhr im Atari (3 Bytes)
SAVMSC	$058/$059	Zeiger auf Video-RAM
VDSLST	$200/$201	Vektor für DLIs
SDMCTL	$22F	Schattenregister DMA-Steuerung
SDLSTL	$230/$231	Zeiger auf Display-List
STICK0	$278	Joystick Buchse 1
STICK1	$279	Joystick Buchse 2
COLOR0	$2C4	Schattenregister Farbe 0
COLOR1	$2C5	-"- Farbe 1
COLOR2	$2C6	-"- Farbe 2
COLOR3	$2C7	-"- Farbe 3
COLOR4	$2C8	-"- Hintergrund
MEMLO	$2E7/$2E8	Zeiger auf Beginn User-Speicher
RUNAD	$2E0/$2E1	Startadresse von Binary-Load Files
INITAD	$2E2/$233	Init-Adresse von

Hardware-Register

COLPF0	$D016	Hardware-Register Farbe 0
COLPF1	$D017	-"- Farbe 1
COLPF2	$D018	-"- Farbe 2
COLPF3	$D019	-"- Farbe 3
COLBK	$D01A	-"- Hintergrund
GRACTL	$D01D	Graphik-Steuerung 3:PM-Graphik an
CONSOL	$D01F	Abfrage Funktionstasten Bit 1-START
WSYNC	$D40A	Stoppt Prozessor bis Ende der Rasterzeile
VCOUNT	$D40B	Momentane Rasterzeile/2

Betriebssystem:

CIOV	$E456	Einsprung 'zentrale Ein-/Ausgabe'
SETVBV	$E45C	Veränderung von Interrupt-Vektoren
SYSVBV	$E45F	Beendet einen 'immediate' VBI
XITVBV	$E462	Beendet einen 'deferred' VBI

ANHANG D: DATGEN

Nützliches Utility zum Umwandeln von ATMAS-II Binärfiles in BASIC-Loader (s. Teil III, Abschnitt 2.5.3)

```
10 REM * DATGEN: Binaer-File in BASIC-Loader umwandeln
30 DIM F$(16),F1$(16)
40 ZLAENGE=65
100 ? "Binaer-File (D:FN.EXT)";:INPUT F$
110 ? "List-File (D:FN.EXT)";:INPUT F1$
115 ? "Anfangs Zeilennummer ";:INPUT Z
120 OPEN #3,4,0,F$:OPEN #4,8,0,F1$
130 GET #3,X1:GET #3,X2:REM * Kennbytes lesen
140 IF X1<>255 OR X2<>255 THEN CLOSE #3:? "Kein Binaer-File!:STOP"
150 GET #3,AL:GET #3,AH:ANFADR=AL+AH*256:REM * Anfangsadresse
160 GET #3,AL:GET #3,AH:ENADR=AL+AH*256:REM * Endadresse
200 REM * BASIC-Laderoutine mit Pruefsumme
210 ? #4;Z;" REM * Binaer-File laden"
220 ? #4;Z+10;" S=0:RESTORE ";Z+100
230 ? #4;Z+20;" FORA=";ANFADR;"TO";ENADR;":READ D:POKEA,D:S=S+D:NEXT A"
240 ? #4;Z+90;" RETURN"
300 REM * Maschinenprogramm als DATA schreiben
310 ZN=Z+100:GOSUB 1000
320 S=0:FOR A=ANFADR TO ENADR:GET #3,D:S=S+D:? #4;D;
330 L=L+LEN(STR$(D))+1:IF L>ZLAENGE THEN ? #4:GOSUB 1000:GOTO 390
340 IF A<>ENADR THEN ? #4;",";
390 NEXT A:? #4
400 ? #4;Z+30;" IF S<>";S;"THEN ?";CHR$(34);"DATEN-FEHLER!";CHR$(34);":STOP"
500 CLOSE #3:CLOSE #4:END
1000 REM * Zeilennr., DATA-Befehl ausgeben
1010 ? #4;ZN;" DATA ";:L=LEN(STR$(ZN))+6:ZN=ZN+10:RETURN
```

ANHANG E: Assembler-Vokabular

Die nachfolgende Zusammenstellung soll Ihnen beim Lesen des Buches eine Hilfestellung bieten. Sollten Sie auf noch unbekannte Begriffe stoßen, dann schlagen Sie hier nach!

Assembler
1) Programm zur übersetzung von -> Quelltext in -> Objektcode.
2) Programmiersprache, die die Befehle des Prozessors als -> Mnemonics enthält.

BCD
Spezieller Code, mit dem dezimale Ziffern im -> Binärsystem verschlüsselt werden. Je vier Binärstellen enthalten eine dezimale Ziffer. Der 6502 kann mit BCD-Zahlen rechnen, die zwei BCD-Ziffern in einem Byte enthalten.

Betriebssystem
Programm, beim Atari im ROM untergebracht, das häufig gebrauchte Routinen (z. B. zur Ein-/Ausgabe) enthält. Wesentlicher Bestandteil ist eine sog. Power-Up-Routine, die den Computer nach dem Einschalten vorbereitet (-> Bootvorgang)

Binärsystem
Zahlensystem, das die '2' als Basis verwendet. Es sind nur zwei Ziffern (0,1) zugelassen. Binärzahlen werden mit einem führenden Prozent-Zeichen gekennzeichnet (z.B. %10010111)

Bootvorgang
Nach dem Einschalten versucht der Computer ein bestimmtes Programm von externen Speichermedien einzulesen (z.B. das DOS von Diskette). Man spricht hier vom 'Booten' des Computers.

Bus: Datenbus, Adressbus
Eine Anzahl paralleler elektrischer Verbindungen zwischen Prozessor, Speicher und Peripheriebausteinen. Der Datenbus umfaßt beim Atari 8 Leitungen, der Adressbus 16 Leitungen.

Carry:
überlauf, der bei Additionen auftreten kann. Beim 6502 wird dieser im 'Carry'-Flag signalisiert.

CIO: Central Input/Output
Routine des -> Betriebssystemes, die nahezu alle Aufgaben zur Ein- und Ausgabe von Daten erledigt. Einsprung: CIOV ($E456)

CPU
'Central Processing Unit', anderer Ausdruck für ->Prozessor.

Direktive
'Pseudo-Befehl' des Assemblers, der zur Steuerung der Assemblierung oder zum Einfügen von Daten in das Objektprogramm dient.

Disassembler
Meist Bestandteil des -> Monitors. Programm, das ein -> Objektprogramm wieder in -> Mnemonics zurückübersetzt. Die Rückübersetzung ist nicht einfach zu beherrschen, da Daten und Programme nicht eindeutig unterschieden werden können und außerdem wichtige Zusatzinformationen wie -> Labels und Kommentare verloren gehen. Vom Disassembler erzeugte Listings sind immer mit Vorsicht zu geniessen!

DMA: 'Direct Memory Access'
Ein Vorgang, bei dem ein Peripheriebaustein ohne Mitwirkung des Prozessors Daten aus dem Speicher liest. Beim ATARI wird Display-List und Video-RAM auf diese Art vom Video-Chip 'ANTIC' gelesen.

Hexadezimalsystem (Abk. Hex)
Zahlensystem zur Basis 16. Verwendet werden 16 Ziffern (0-9 und A-F). Das hexadezimale Zahlensystem eignet sich zur Darstellung von 8 und 16-Bit langen -> Binärzahlen, die damit wesentlich 'handlicher' im Umgang werden. 'Hex-Zahlen' werden mit einem führenden Dollar-Zeichen markiert (z.B. $E456).

Immediate
Engl. für 'unmittelbar', bezeichnet beim 6502 eine Adressierungsart, die den -> Operanden als Konstante (Datum) auffaßt.

Interrupt
Bei bestimmten Anlässen (etwa einem Tastendruck) kann der Prozessor sein momentanes Programm verlassen und eine sog. Interrupt-Routine bearbeiten. Auf diese Weise können äußere Vorgänge Reaktionen im Computer hervorrufen.

Label
Eine Hilfestellung des -> Assemblers, mit der Adressen und Daten mit symbolischen Namen belegt werden können. Damit können sinnvolle Bezeichnungen anstatt Zahlenwerten im Programm verwendet werden. Dies erhöht die Lesbarkeit von Programmen und erleichtert die Programmierarbeit. Beim Assemblieren werden Labels durch ihre Zahlenwerte ersetzt.

LSB
'Least Significant Byte', bezeichnet das niederwertige Byte eines -> Wortes. Das LSB von $E456 ist $56.

Makro
Ein Makro-Befehl setzt sich aus mehreren Elementar-Befehlen zusammen. Auf diese Weise kann man sich neue Befehle für immer wiederkehrende Befehlsfolgen schaffen, die aus mehreren Einzelbefehlen aufgebaut sind.

Maske
Eine Maske ist eine meist -> binär dargestellte Zahl, die mit anderen Zahlen logisch verknüpft wird. Je nach Verknüpfung und Daten werden somit einzelne Bits 'maskiert'.

Mnemonic
Darunter versteht man Kürzel für Assemblerbefehle (z.B. LDA, STA). Der -> Assembler übersetzt die Mnemonics in Codes, die der Prozessor ausführen kann.

Monitor
Programm zur Fehlersuche in Maschinenprogrammen. Oft auch 'Debugger' genannt.

MSB
'Most Significant Byte', bezeichnet das höherwertige Byte eines -> Wortes. Das MSB von $E456 ist $E4.

Objektcode
Vom -> Assembler erzeugtes, lauffähiges Maschinenprogramm.

Op-Code
1) Code eines Assemblerbefehles (Angabe meist hexadezimal)
2) Assemblerbefehle

Operand
Zusätzliche Information bei Assemblerbefehlen. Zumeist Adressen, auf die sich die Befehle beziehen sollen.

Page
Bezeichnet einen zusammenhängenden Bereich von 256 Bytes, die an einer 'Pagegrenze' beginnen. Eine Pagegrenze ist eine Adresse, die durch 256 teilbar ist.

Prozessor
Zentrale Verarbeitungseinheit (-> CPU), in der alle Vorgänge gesteuert werden und alle Rechenoperationen ablaufen. Der Prozessor enthält -> Register zur Ablaufsteuerung und zur Zwischenspeicherung der Ergebnisse. Er ist über -> Bus-Systeme mit Speicher und Peripheriebausteinen verbunden.

Quellprogramm
Assemblerprogramme werden mit Hilfe eines Texteditors eingegeben. Dabei werden Kürzel für Assemblerbefehle (-> Mnemonics) und -> Labels verwendet. Der Assembler übersetzt diesen Text in das -> Objektprogramm.

Register
1) Spezielle Speicherzellen innerhalb des -> Prozessors (Prozessor-Register, Akku, Index-Register).
2) Häufig auch Bezeichnung für beliebige Speicherzellen, die zu Berechnungen im Programm hergenommen werden, oft Bezeichnung für Speicherzellen, die zu Ein-/Ausgabezwecken dienen (Hardware-Register).

Stack
Speicherbereich von $0100 bis $01FF, der hauptsächlich zum Aufbewahren von Rücksprungadressen verwendet wird. Die Verwaltung des Stacks übernimmt ein -> Register des Prozessors (Stack-Pointer).

Vektor
Zwei Speicherzellen, die die Adresse einer anderen Speicherzelle beinhalten. Oft auch als Zeiger oder Pointer bezeichnet, da man sich bildlich vorstellen kann, daß der Vektor auf eine andere Speicherzelle zeigt. Gelegentlich benennt man auch einen JMP-Befehl (in einer Sprunqtabelle) als Vektor, z.B. den CI0-Vektor bei $E456.

'VBI:Vertical Blank Interrupt
Ein -> Interrupt, der nach jeder Erzeugung eines Video- Bildes ausgelöst wird (50 mal pro Sekunde). Mittels zweier Vektoren kann man eigene Programme in den VBI einbinden. Ändert man die Graphik nur im VBI, so kann man Störungen und Flimmern verhindern.

Video-Speicher
Korrekte Bezeichnung ist Bildwiederholspeicher, hier ist die am Bildschirm dargestellte Information im Speicher abgelegt. Beim Atari ist das 'Video-RAM' nicht an eine feste Adresse gebunden, sondern kann an beliebigen Stellen angelegt werden.

Wort
Als Wort (engl. Word) bezeichnet man eine Zahl, die zur Erweiterung des Zahlenbereiches aus zwei Bytes zusammengesetzt ist (z.B. zur Darstellung von Adressen). Beim 6502 wird immer zuerst das ->LSB, dann das ->MSB im Speicher abgelegt.

Zero-Page
Vom Prozessor bevorzugt behandelter Bereich, der besonders schnell bearbeitet werden kann. Besondere Adressierungsarten sind nur dort zulässig. Die Zero-Page enthält 256 Bytes von Adresse $0000 bis $00FF.

Hier ist sie: Die leichtverständliche Einführung in die Assemblerprogrammierung Ihres ATARI-Computers. Sie brauchen keinerlei Vorkenntnisse in Assembler, wenngleich ein paar Erfahrungen in BASIC schon recht hilfreich wären.

Sie erfahren ganz genau, wie man in Assembler programmiert, was Direktiven zu bedeuten haben und wozu ein Maschinensprache-Monitor eigentlich gut ist. Der Schwerpunkt dieses Buches liegt dabei eindeutig auf der Anwendung, d.h. Sie lernen keine trockenen Theorien, sondern sehen immer gleich ein Beispiel, das Sie sofort abtippen können. Und das hat seinen Grund, denn gerade bei der Programmierung in Assembler macht nur die Übung den Meister.

Als Handwerkszeug zu diesem Buch empfiehlt sich der ATMAS-II Makroassembler, da die Beispiele speziell auf dieses Software-Paket zugeschnitten sind.

DAS ASSEMBLERBUCH kann mit allen 8-Bit ATARI-Computern verwendet werden (600XL(64K)/800XL und 130XE, sowie 400/800 ab 48K).

Der Autor:

Peter Finzel, Jahrgang 1959, ist Ingenieur der Elektrotechnik und beschäftigt sich schon seit langem mit der Programmierung von ATARI-Computern. Aus seiner Feder stammen Programme wie CAVELORD und SCHRECKENSTEIN. Neben seinem ersten Buch, der 1984 erschienenen 'HEXENKÜCHE', ist er Autor der ASSEMBLERECKE, einer Artikelserie, die seit Februar 1985 regelmäßig in der Zeitschrift 'Computer-Kontakt' zu lesen ist.

Finzel: Das Assemblerbuch für ATARI®-Computer