Inhalt

Assembler ist keine Alchimie

Den kompletten Assembler-Kurs in einem Stück wünschten sich viele 64'er-Leser. In diesem Sonderheft können wir diesen Wunsch realisieren. Der Kurs soll nicht unbedingt ein Buch über Maschinensprache ersetzen, er wird Ihnen jedoch helfen, diese Sprache leichter zu verstehen.

Vermutlich hat es Ihnen auch schon ab und zu in den Fingern gejuckt, wenn Sie von Wunderdingen gelesen haben, die man per Maschinensprache mit dem Computer machen kann. Vielleicht haben Sie sogar schon mal nichtsahnend angefangen einzutippen, was Sie als Assemblerlisting sahen. Doch schon nach dem ersten »C000 LDA # $00« und RETURN weigerte sich der Computer mit einem lapidaren »SYNTAX ERROR«. Wieso, werden Sie sich gefragt haben, das ist doch nun die Sprache unserer Maschine, nämlich Maschinensprache, was habe ich falsch gemacht?

Dann sind Sie sicherlich mal auf diese merkwürdigen Basic-Programme gestoßen, in denen ein langer Wurm von DATA-Zeilen mit einem kleinen FOR..NEXT.. POKE-Kopf vorne und einem SYS-Schwanz hinten enthalten ist, und die man Basic-Lader nennt. Sie haben fleißig Zahlen eingetippt – das Ganze hoffentlich sofort gespeichert-, vorschriftsmäßig mit dem SYS-Befehl gestartet und auf einen scheintoten Computer geschaut, der nur noch durch Aus- und Einschalten wiederzubeleben war. Wenn Sie dann nach langer Fehlersuche den irrtümlich eingetippten Punkt durch ein Komma ersetzt haben (oft finden Sie auch keinen Fehler, denn bei langen DATA-Sequenzen schlägt der Druckfehlerteufel mit Vorliebe zu), werden Sie sich gefragt haben, warum in aller Welt dieses kleine Mißgeschick den ganzen Computer abstürzen läßt. Sie merken vermutlich schon, daß mir das alles und noch mehr (worüber ich schamhaft schweige) passiert ist. Die Konsequenz war, daß ich losging, um ein schlaues Buch zu erwerben. Aber merkwürdig, damals tauchte der Begriff »Maschinensprache« in keinem Titel auf. Irgendwann begriff ich, daß Assembler und Maschinensprache irgend etwas miteinander zu tun haben.

Aber da fing das ganze Elend erst richtig an: Da gab es 6502-, Z80-, 8080-, 8085-, 6800-Assembler, da waren irgendwelche Schaltpläne, anscheinend, wie man wo was hinlötet- für mich als Nichtelektroniker eine Art moderner Kunst-, da war von CPU, Bussen, negativen Flanken, Zweiphasentakten die Rede.

Ich habe mich furchtbar geärgert über die Geheimsprache, die es dem Uneingeweihten verwehrt, etwas zu verstehen. Seither hat sich einiges verändert. Die Geheimnisse sind keine mehr und ich werde Ihnen in dieser Serie ohne verschlüsselte Sprache die magischen Zirkel der Assembler-Alchimisten offenbaren. Heute gibt es auch Bücher über »Maschinensprache auf dem Commodore 64« und es sei Ihnen angeraten, ruhig auch das eine oder andere durchzuarbeiten. Sie werden allerdings feststellen, daß die meisten davon gerade dort aufhören, wo es anfängt spannend zu werden: bei der Benutzung von Routinen des Betriebssystems

und des Interpreters. Deswegen soll der Schwerpunkt dieses Artikels woanders liegen:

Wir werden das notwendige Grundwissen über die Hardware nur ganz knapp behandeln, dann das Vokabular des 65xx-Assemblers kennenlernen. Den Hauptteil des Artikels verbringen wir aber mit Dingen, über die es kaum Literatur gibt, nämlich wie man für eine Unzahl von Programmieraufgaben nicht nochmal das Rad erfinden muß, weil es schon längst in unserem Computer existiert.

Bevor wir loslegen, will ich Ihnen noch etwas Literatur empfehlen:

a) Wenn wir über Speicheraufbau, das binäre und das hexadezimale Zahlensystem reden, sollten Sie die Serie »Reise durch das Wunderland der Grafik« gelesen haben, die in der 64'er in den Folgen 1 und 2 (Ausgaben 4/84 und 5/84) diese Themen grundlegend behandelt hat. (Auch als Buch unter gleichnamigem Titel erschienen.)

b) Als Nachschlagebuch sehr wertvoll ist das Buch von Raeto West: C 64 Computer Handbuch. Hier finden Sie auch viele Tips und Tricks.

c) Später wird Ihnen dieses Buch fast unentbehrlich vorkommen: R. Babel, M. Krause, A. Dripke: Systemhandbuch zum Commodore 64 (und VC 20), München 1983

Weitere Literaturempfehlungen werde ich Ihnen von Fall zu Fall geben und Sie finden sie auch in der Bücherecke. Gerade zu unserem Computer erscheint fast jeden Monat ein neues Buch und es ist nicht einfach, die Spreu vom Weizen zu trennen.

1. Einige Begriffsklärungen

Zunächst einmal muß ich Sie enttäuschen: Ich glaube kaum, daß Sie mit Ihrem Computer je einmal in Maschinensprache verkehren werden! Maschinensprache, das ist die einzige, die der Computer direkt versteht, das sind vorhandene oder nicht vorhandene Stromimpulse oder Magnetisierungszustände, die bei unserem Computer durch 8-Bit-Binärzahlen auszudrücken sind. Was wir mit unserem Computer reden werden ist Assembler. Mit dem Computer sprechen soll heißen: Mit dem Gehirn unseres Computers, dem Prozessor, oft auch CPU (von Central Processing Unit=Zentraler Arbeitsbaustein) genannt, verkehren, also ihm Befehle zu geben. Solche CPUs werden bei verschiedenen Firmen hergestellt, sind daher unterschiedlich aufgebaut und auch unterschiedlich ansprechbar. Ein weit verbreiteter Prozessortyp ist der 6502, der das Gehirn des C 64 und auch des VC 20 ist. Genau genommen ist das Gehirn des C 64 allerdings der 6510, ein dem 6502 fast identischer Prozessor. Auf den kleinen Unterschied werden wir noch zu sprechen kommen. Beide (6502 und 6510) sind in 6502-Assembler zu programmieren und wenn wir diese Sprache sprechen, sind für uns alle 6502-Computer zugänglich: Commodore, Apple, Atari und einige andere. Nun wissen Sie aber immer noch nicht, was Assembler eigentlich ist. Das englische Wort »assemble« heißt auf deutsch etwa montieren, zusammenstellen. Es handelt sich also um eine Programmiersprache und weil sie sehr eng am Computer orientiert ist, spricht man von einer »maschinenorientierten« Programmsprache im Gegensatz zu »problemorientierten« Programmsprachen wie Basic, Pascal, Cobol etc., die – so sollte es jedenfalls sein – auf jedem Computertyp gleich aussehen.

Ein Assembler ist aber noch etwas anderes, nämlich ein Software-Instrument, das einen in Assembler geschriebenen Befehl in die Maschinensprache übersetzt. Man spricht vom Vorgang des Assemblierens. Das umgekehrte leistet ein Disassembler, welcher uns Maschinensprache durch Rückübersetzung lesen hilft. Um die Verwirrung noch etwas zu steigern, sage ich Ihnen auch noch, was ein Monitor ist. In diesem Zusammenhang ist kein Bildschirmgerät damit gemeint, sondern ebenfalls ein Software-Instrument, das den Einblick in die Register und Speicher des Computers gewährt.

Damit Sie nun den Überblick völlig verlieren, sei abschließend zu diesem Sprachenwirrwarr noch erzählt, daß Software-Pakete, die sowohl Assembler als auch Disassembler als auch Monitor enthalten und noch eine Menge anderer brauchbarer Dinge, oft als »Assembler« angeboten werden. Das ist ein alter Trick der Alchimisten, verschiedenen Dingen den gleichen Namen zu geben!

2. Basic contra Assembler

Um das Nachfolgende deutlich zu machen, schalten Sie bitte Ihren Computer an und tippen die beiden folgenden Programme ein, die beide genau dasselbe tun: Das obere Viertel unseres Bildschirmes mit dem Buchstaben A füllen (beim VC 20 ist es die obere Hälfte). Zunächst einmal in Basic:

```
10 FOR I=1024+255 TO 1024 STEP-1
20 POKE I, 1:POKE I+54272,14
30 NEXT I
```

Für den VC 20 (Grundversion und 3-KByte-Erweiterung) ist zu setzen: anstelle von 1024 jetzt 7680, statt 54272 jetzt 30208 und statt 14 die 6. Wenn Sie mehr als die 6,5 KByte im VC 20 haben, dann setzen Sie statt 1024 jetzt 4096, statt 54272 jetzt 34304 und ebenfalls statt 14 die 6. Das Programm braucht 55 Byte + 7 Byte für die Variable I, macht zusammen 62 Byte Speicherplatz. Es geht ganz schnell und wenn Sie es schaffen, können Sie ja mal mitstoppen, wie lange es von RUN bis READY braucht: zirka 4 Sekunden.

Jetzt dasselbe in Assembler. Weil wir aber noch nicht soweit sind, erst mal als Basic-Lader, der uns das Programm in den Speicher bringt (wir kommen dazu gleich noch). Geben Sie also NEW ein und dann:

```
10 FOR I=7000 TO 7000+16
20 READ A:POKE I,A:NEXT I :END
30 DATA 160,255,162,14,169,1,153,255,
3,138,153,255,215,136,208,244,96
```

Beim VC 20 geben Sie bitte statt der 14 (Zeile 30,4.Zahl) eine 6 ein. Starten Sie den Basic-Lader mit RUN und nach dem READY geben Sie NEW und CLR ein: wir brauchen ihn nicht mehr. Ab Speicherstelle 7000 steht jetzt unser Assemblerprogramm als Maschinencode. Daß es wirklich dasselbe tut wie das Basic-Programm erfahren Sie durch SYS 7000. Da hatten Sie vermutlich gar keine Zeit mehr, auf die Stoppuhr zu drücken! (5,4 Millisekunden etwa dauert das ohne die Zeit, die der Basic-Interpreter für den Befehl SYS benötigt). Außerdem braucht das Programm 17 Byte Speicherplatz.

Genau das ist es, was die Assemblerprogrammierung so reizvoll macht: Der Speicher faßt mehr an Programm und die Ausführung des Programmes geht fast 1000mal so schnell! Dazu kommen natürlich noch einige andere Kriterien, denn viele Probleme sind zum Beispiel in Basic nicht lösbar, sondern nur mit dem vielseitigeren Assembler.

Unser Computer ist darauf vorbereitet, daß wir ihn in Basic ansprechen. Er enthält im Normalfall sofort nach dem Einschalten ein stets präsentes Übersetzungsprogramm, den Interpreter, welcher unsere Basicanweisungen für ihn verständlich interpretiert. Auch das ist ein Unterschied zu Assemblerprbgrammen: Ist ein solches Programm erst einmal assembliert (also als Maschinensprache im Speicher vorhanden), braucht man kein Übersetzungsprogramm mehr. Basic-Programme dagegen müssen bei jedem Durchlauf von vorne bis hinten ständig übersetzt werden, sie laufen nicht ohne vorhandenen Interpreter. Wie so ein Interpreter im Prinzip arbeitet und was ihn von einem sogenannten Compiler

unterscheidet, können Sie im 64'er, Ausgabe 4/84 und im 64'er Sonderheft 6 (Top-Themen) im Artikel von M. Törk über seinen Strubs-Precompiler nachlesen.

Dort sehen Sie dann auch, daß ein Compiler zwar ein Basic-Programm enorm beschleunigen kann, aber bei weitem nicht an die Geschwindigkeit reiner Assemblerprogramme heranreicht, vom Speicherplatzbedarf ganz zu schweigen.

3. Wie sag ich's meinem Computer?

Leider haben weder der C64 noch der VC 20 einen Assembler implementiert. (Sie merken, daß jetzt von dem Software-Paket die Rede ist!). Es gibt einen etwas mühseligen Weg, dieses Handicap zu umgehen: den Basic-Lader. Wie ist also der Weg, mit einem solchen Lader eigene Maschinenprogramme in den Computer zu bekommen?

a) Erstellen des Assemblerprogrammes. Das zu lernen ist die Hauptaufgabe in diesem Artikel. Das Ergebnis wird eine Kette von Befehlen sein, zu denen zum Beispiel der Befehl RTS gehört.

b) Jedem Befehl in Assembler entspricht in Maschinensprache ein Binärcode in einer Speicherstelle. Diese Codes sind in Listen nachschlagbar: RTS entspricht dem Binärcode 0110 0000.

c) Der Code muß in eine Speicherstelle eingegeben werden. Das geschieht von Basic aus mit dem POKE-Befehl. Weil aber Basic keine Binärzahlen kennt, muß der Code ins Dezimalsystem umgerechnet werden. Glücklicherweise sind in den Tabellen meist schon die Codes als Dezimal- oder wenigstens als Hexadezimalzahlen enthalten. RTS ist dezimal 96 (oder hexadezimal 60, das auch $ 60 geschrieben werden kann). Man POKEt nun an die richtige Adresse den Wert 96, also zum Beispiel POKE 7016,96

d) Auf diese Weise wird Byte für Byte in der Programmabfolge verfahren. Das reine POKEn geschieht dann eben in der Form wie im oben gezeigten Basic-Lader. Mühsam, mühsam! Auch kann man leider nur mit dem PEEK-Kommando nachsehen, was denn nun im Speicher steht (PEEK (7016) gibt uns den Wert 96, entsprechend RTS).

Ein anderer Weg ist, den in diesem Sonderheft abgedruckten »SMON« abzutippen, oder sich die Leser-Service-Diskette zu bestellen.

Assembler (das Software-Paket) gibt es in den unterschiedlichsten Ausführungen. Es gibt beispielsweise Direkt-Assembler, die jede Programmzeile sofort nach dem RETURN assemblieren, aber auch 2-Pass-Assembler, bei denen das erst nach Abschluß des Programms insgesamt durch einen Befehl (zum Beispiel ASSEMBLE) geschieht. Bei einigen kann man (ähnlich wie bei Basic mit REM) Kommentare anfügen, bestimmten Programmstellen Namen geben (LABEL), ganze Programmabschnitte mit einem Merknamen aufrufen (MAKROS) und so weiter. Was Sie für sich bevorzugen, bleibt Ihnen natürlich überlassen. Die in diesem Artikel beschriebenen Programme werden am Anfang auf diese schönen Erleichterungen verzichten, es wird sozusagen der nackte Assembler verwendet. Was Sie aber außer dem reinen Assembler noch brauchen, ist ein Disassembler und ein Monitor (ich habe schon erklärt, welchen ich meine), damit wir unseren Computer (fast) immer im Griff haben.

4. Wie funktioniert unser Computer?

Weil das Programmieren in Assembler einen viel engeren Kontakt zu technischen Einzelheiten unseres Computers erfordert, ist es notwendig, ein wenig über diese Innereien und ihre Funktion zu wissen. Sehen Sie sich dazu bitte das Bild 1 an.

Da sehen wir zunächst unseren Mikroprozessor, der meist eine Menge Funktionen in sich vereinigt (dazu kommen wir noch). Im Prinzip ist das unsere CPU (Zentraler Arbeitsbaustein). Der Prozessor steht über eine Reihe von Leitungen mit dem Rest des Computers in Verbindung. Diese Leitungen werden im Fachjargon BUSSE genannt. Da ist zunächst einmal der sogenannte Adreßbus, auf dem 16-Bit-Adressen transportiert werden, die der Prozessor erzeugt, und die die Herkunft oder auch das Ziel von Daten festlegen, die über den Datenbus laufen. Dieser kann 8-Bit-Daten transportieren, und zwar schreibend oder lesend, also zum Beispiel vom Prozessor zum RAM (schreibend), vom RAM zum Prozessor (lesend) und so weiter. Außerdem gibt es da noch einen Steuerbus, der verschiedene Synchronisationsaufgaben durchführen hilft. Links vom Prozessor ist ein Taktgeber angedeutet. Damit nichts durcheinander kommt, läuft alles im Computer sozusagen im Gleichschritt. Diese Uhr ist gewissermaßen der Trommler, den Sie vielleicht von den alten Ruder-Galeeren kennen. Dann sehen Sie einen ROM-Bereich, also einen Nur-Lese-Speicher (Read Only Memory). Daß man hier nur herauslesen kann, ist durch den Pfeil zum Datenbus

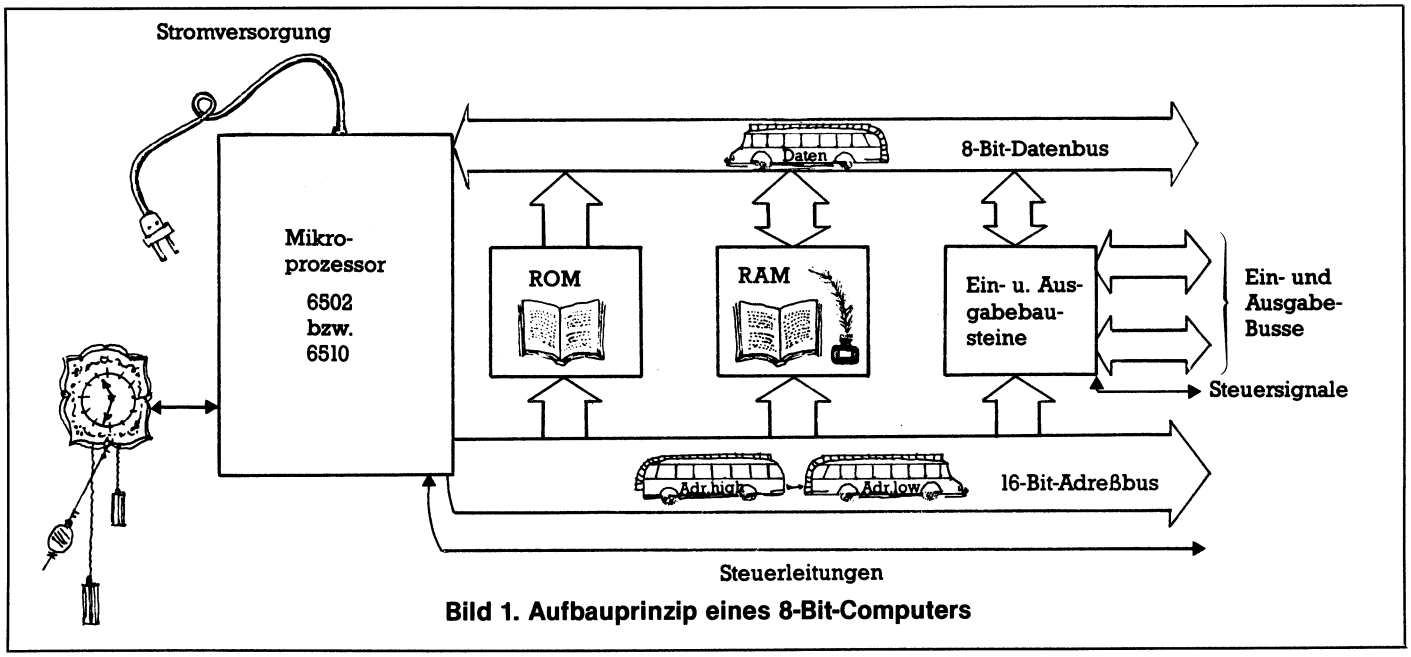


Bild 1. Aufbauprinzip eines 8-Bit-Computers

gekennzeichnet. Doppelpfeile finden wir aber beim RAM (Random Access Memory), einem Speicher für beliebigen Zugriff, also lesend und schreibend, und bei den Ein- und Ausgabebausteinen, die den Kontakt des Computers mit der übrigen Welt erlauben, also auch mit uns. Dieses Aufbauprinzip finden wir bei allen 8-Bit-Computern.

5. Das Innenleben eines Mikroprozessors

Um es gleich nochmal zu sagen: Was hier erzählt wird, ist nicht dazu geeignet, Elektronik-Freaks den totalen Durchblick zu geben. Wenn Sie das aber gerne möchten, dann sehen Sie sich zum Beispiel die Blockschaltbilder an im »Programmer's Reference Guide« für den Commodore 64 auf Seite 404 oder im »MOS-Hardware-Handbuch« auf Seite 34. Auch Rodney Zaks' Buch »Programmierung des 6502« ist zu empfehlen. Er hat sich viel Mühe gegeben, sich verständlich auszudrücken. Mir kommt es nur auf den allgemeinen Überblick an. Den sollen Sie bekommen, wenn wir uns jetzt zusammen Bild 2 betrachten.

Da sehen Sie zunächst als Herzstück des Prozessors, die ALU (Arithmetik Logical Unit), also den arithmetisch-logischen Baustein. Die ALU hat die Fähigkeit, Rechenoperationen auszuführen mit Daten, die sie über den Datenbus und normalerweise vom Akkumulator erhält. Das Ergebnis wird ebenfalls im Akkumulator abgelegt (daher auch der Name: von akkumulieren, etwas ansammeln). Der Akkumulator ist das Register, das uns als Programmierer am häufigsten beschäftigen wird. Er ist die Sammel- aber auch die Verteilerstelle für fast alle Daten, die wir hin- und herschieben wollen. Sowohl der Akku (so werde ich ihn, in der Hoffnung auf Ihr wohlwollendes Verständnis, künftig bezeichnen) als auch alle anderen Register, das heißt, die höchste Zahl, die darin bearbeitet werden kann, ist 255 (binär 1111 1111). Nahezu ebenso oft wie den Akku werden wir die beiden sogenannten Index-Register X und Y benutzen. Warum man sie Index-Register nennt, werden Sie noch im Verlauf des Kurses sehen. Als nächstes zum Prozessor-Statusflaggen-Register (hier P genannt). Man findet darin angezeigt, ob eine Rechenoperation ein negatives Ergebnis hatte, ob eine Null aufgetaucht ist oder ob ein Übertrag stattgefunden hat. Auch dieses Register wird uns noch häufig begegnen. Das Stapelregister, auch Stackpointer (Stapelzeiger) genannt, gibt uns Auskunft über den Füllungsgrad eines 256 Byte großen speziellen Speichers, der vom Prozessor direkt verwaltet wird. Auch damit werden wir noch oft zu tun haben. Schließlich kommen wir zur vorhin erwähnten Ausnahme, zum Programmzähler (PCL, PCH). Das ist ein 16-Bit-Register, das sich aus zwei 8-Bit-Registern (PCL für das LSB und PCH für das MSB) zusammensetzt und daher alle 65535 Speicherplätze ansprechen kann. Hier ist immer die Adresse des nächsten abzuarbeitenden Befehls enthalten.

Ich will an dieser Stelle nicht in die Einzelheiten der Befehlsabarbeitung einsteigen (das können Sie auch bei Rodney Zaks nachlesen, wenn Sie es genau wissen wollen). Es soll nur gesagt sein, daß sich die Verarbeitung in drei Schritte unterteilen läßt:

a) den nächsten Befehl holen
b) den Befehl decodieren
c) den Befehl ausführen

Zu c) ist noch zu sagen, daß es Befehle gibt, die der Prozessor ohne weitere Angaben ausführen kann. Für andere müssen erst noch weitere Daten aus dem Speicher geholt oder dort abgelegt werden. Deswegen brauchen die Befehle unterschiedliche Zeiten zur Ausführung. Die Zeit wird als Anzahl von sogenannten Taktzyklen in den Befehlstabellen angegeben. Unser Computer hat eine Taktfrequenz von rund 1 MHz, was bedeutet, daß ein Taktzyklus etwa eine Mikrosekunde (10^{-6} Sekunden) dauert. Auf diese Weise wurde die Zeitdauer für unser kleines Demonstrationsprogramm zu Anfang berechnet. Auch das werden Sie noch lernen.

6. Der Speicher unseres Computers: eine Straße mit 65536 Hausnummern

Dieser Artikel ist für den VC 20 und den C 64 geschrieben. Den Speicheraufbau des Commodore 64 finden Sie in der April-Ausgabe '84 dieser Zeitschrift ab Seite 119. Deswegen soll hier nur der des VC 20 gezeigt werden. Man muß beim VC 20 zwei Konfigurationen unterscheiden — sehr zum Leidwesen der Benutzer. In Bild 3 ist die Aufteilung gezeigt, die in der Grund- und der um 3 KByte erweiterten Version vorliegt.

In Bild 4 sehen Sie die Speicheraufteilung, die bei mehr als 6,5 KByte eingestecktem Speicher gültig ist.

Wenn Sie die VC 20 Speicherarchitekturen mit der des C64 vergleichen, werden Sie eine Reihe von Unterschieden

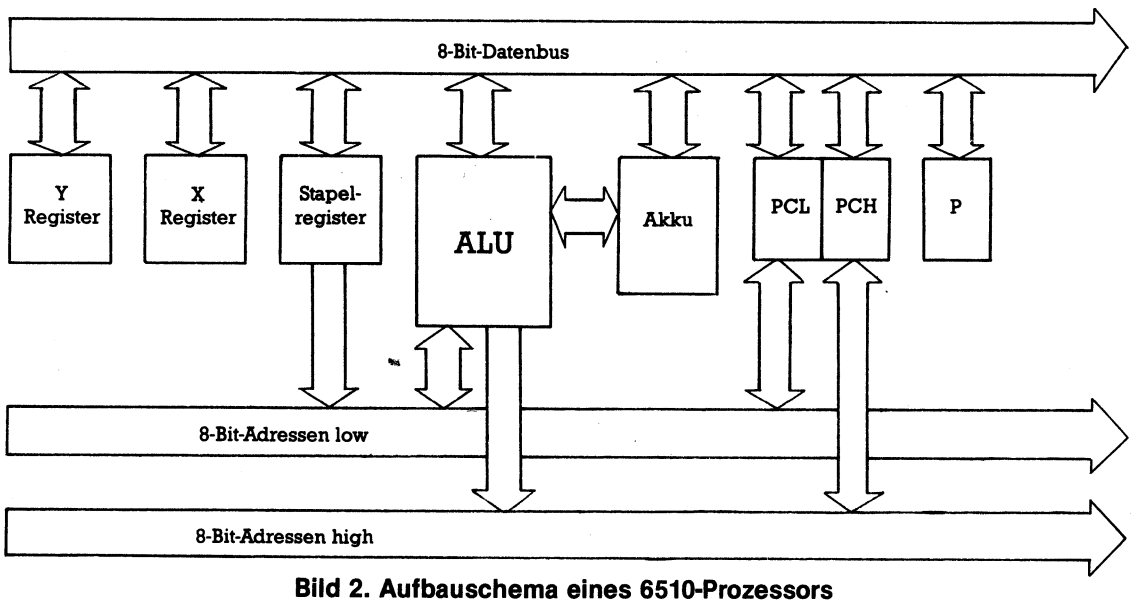


Bild 2. Aufbauschema eines 6510-Prozessors

feststellen. Genau besehen gibt es an den wichtigen Punkten aber eine Menge Gemeinsamkeiten! Der VC 20 kennt nur Speicher-Häuser mit Erdgeschoß, im Gegensatz zum C 64,

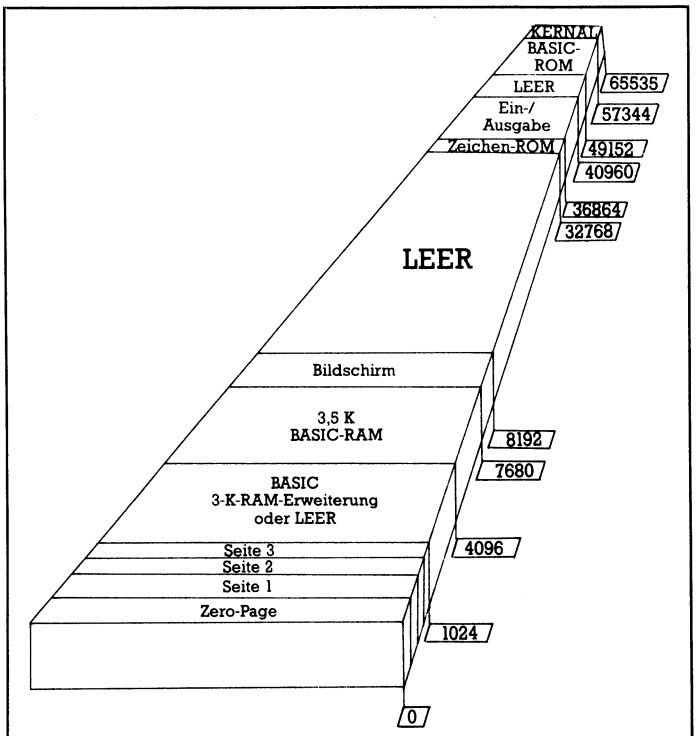


Bild 3. VC 20-Speicher (Grundversion oder mit 3-KByte-Erweiterung)

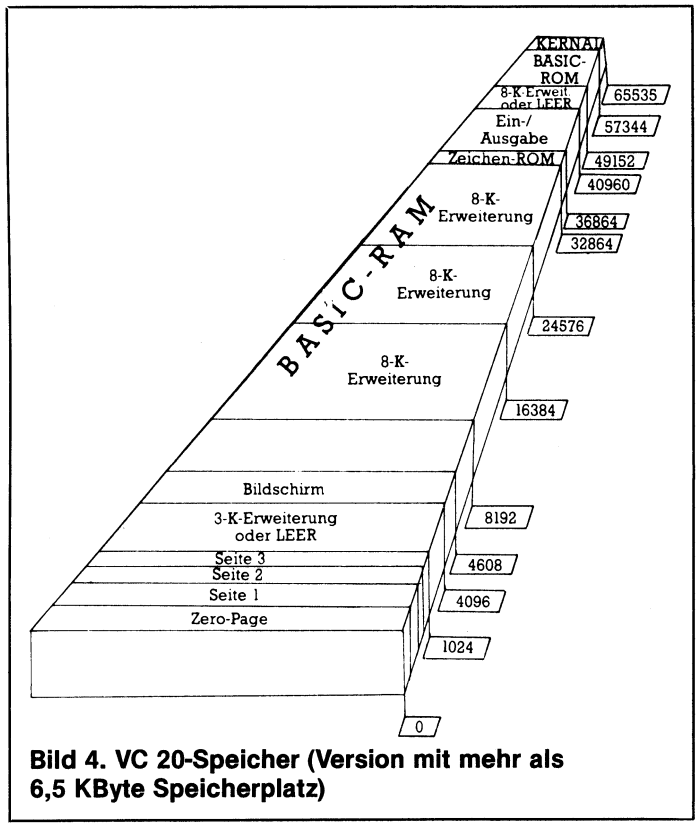


Bild 4. VC 20-Speicher (Version mit mehr als 6,5 KByte Speicherplatz)

wo manche Bereiche sogar zwei Etagen haben (soll heißen: mehrfach belegt sind). Durch die Eigenart des C 64 aber, im Normalfall das Basic-ROM, die Ein- und Ausgabebausteine und das Betriebssystem eingeschaltet zu haben, kann man ihn eigentlich genauso behandeln wie einen VC 20, bei dem die genannten ROM-Bausteine, – und zwar das Basic-ROM —, um 8 KByte verschoben sind. Die Unterschiede der ROM-Inhalte können fast vernachlässigt werden. Wir werden im Einzelfall darauf zu sprechen kommen. Bei den Ein- und Ausgabebausteinen liegen allerdings größere Unterschiede.

Die Seiten 0 bis 3 (eine Seite oder auch page enthält 256 Byte und man zählt oft auch in diesen Seiten, wenn vom Speicher die Rede ist), sind sich ebenfalls sehr ähnlich und die wenigen Unterschiede werden uns ebenfalls noch beschäftigen. Der Bildschirm liegt bei der Grundversion und der mit der 3-KByte-Erweiterung von 7680 bis 8191, in der Version mit mehr als 6,5 KByte von 4096 bis 4607 und beim C 64 von 1024 bis 2047. Der Bildschirmfarbspeicher liegt – bei gleicher Reihenfolge – von 37888 bis 38399, beziehungsweise von 38400 bis 38911 und schließlich von 55296 bis 56295. Der Basic-RAM-Bereich beginnt beim C 64 im Normalfall bei 2048 und endet bei 40959. Beim VC 20 ist das natürlich wieder von der jeweiligen Erweiterung abhängig (Tabelle 1).

Grundversion	:Basic-Start	4096	Basic-Ende	7679
+3-K-Erweiterung	: —"—	1024,	—"—	7679
+8-K-Erweiterung	: —"—	4608,	—"—	16383
+16-K-Erweiterung	: —"—	4608,	—"—	24575
+24-K-Erweiterung	: —"—	4608,	—"—	32767

Tabelle 1. Basic-Start und -Endadressen beim VC 20 mit verschiedenem Speicherausbau

Dies gilt – wie Sie leicht auch aus Bild 4 sehen können – auch dann, wenn zu den 8 KByte/16 KByte/24-KByte-Erweiterungen noch die 3-KByte-Erweiterung und die 'KByte-Erweiterung im hohen Speicherbereich (40960 bis 49151) verwendet werden. Diese letztgenannten Adressenbereiche sind dann gut als geschützte RAM-Bereiche für Maschinensprache zu verwenden, ebenso wie beim C 64 der Speicherabschnitt von 49152 bis 53247.

7. Auskunft über das Befinden unseres Computers: die Register-Anzeige

Bisher haben wir uns mit dem Innenleben unserer Computer auseinandergesetzt und die wichtigsten Teile der Hardware kennengelernt. Jetzt kommen wir zur Software, nämlich zum Assembler. Wenn Sie jetzt den SMON einschalten, meldet er sich mit einer Registeranzeige (Bild 5).

Die angezeigten Werte sind Beispiele, wie sie beim C 64 auftreten können. PC ist der Programmzähler, der immer auf den nächsten zu holenden Befehl zeigt. (Der Wert $E147 rührt vom SYS-Aufruf, mit dem ich meinen Assembler starte). IRQ zeigt uns an, auf welche Adresse der sogenannte Interrupt-Vektor gestellt ist. Das ist das Byte-Paar 788 (LSB) und 789 (MSB). Auf den Wert $EA31 zeigt es im Normalfall.

Die nächsten acht Angaben beziehen sich auf das Prozessorstatusregister, das wir zuletzt P genannt haben. Die Bedeutung der einzelnen »Flaggen« zeigt Ihnen Bild 6.

AC ist der aktuelle Inhalt des Akkus. XR zeigt an, was im X-Register und YR was im Y-Register enthalten ist. SP (von Stackpointer = Stapelzeiger) gibt uns Auskunft über den freien Platz im Stapelregister. Damit wissen wir genau, was in diesem Moment in unserem Computer vorgeht. So fremd Ihnen das alles im Augenblick noch vorkommt, bald werden Sie mit dieser Registeranzeige auf vertrautem Fuß stehen.

8. Wie sieht ein Assemblerprogramm aus?

Das menschliche Gehirn hat dem des Computers vieles voraus. Dazu gehört beispielsweise, daß ein Mensch allerlei Dinge gleichzeitig tun kann: gehen, sprechen, Musik hören, lächeln, Handbewegungen ausführen, womöglich dabei auch noch etwas kauen und so weiter. Ein Computer ist dazu nicht imstande. Er erledigt eine kleine Aufgabe nach der anderen. Weil er das so schnell macht, hat es für uns den Anschein, es geschähe alles gleichzeitig. Das Maschinenprogramm ist eine Kette solcher kleiner Aufgaben. Das erste Glied daraus, das wir kennenlernen wollen ist der Befehl

LDA.

Das bedeutet: Lade den Akkumulator. Alle Assembler-Befehlsworte bestehen aus drei Buchstaben wie dieser hier auch. Wir haben in der ersten Folge schon gesagt, daß einem solchen Befehl eine 8-Bit-Codezahl entspricht. Das ist hier $A9 oder binär 1010 1001 oder schließlich dezimal 169. Die Codezahl muß in einem Speicherplatz stehen, zum Beispiel in $1500 (entspricht dez. 5376). Assemblerlistings sehen dann so aus:

1500 LDA

Hier tritt also die Speicherplatznummer mit einem nachfolgenden Befehl anstelle der von Basic gewohnten Zeilennummer.

Es fehlt noch etwas Entscheidendes: Was soll denn in den Akku geladen werden? Genauso wie es in Basic Befehle gibt, die für sich alleine stehen können wie CLR oder LIST, gibt es auch im Assembler solche Befehle. Weitaus häufiger sind aber hier Befehle, die ein Argument erfordern (in Basic zum Beispiel PEEK(100). Dabei ist 100 das Argument). In Assembler gibt es zwei Sorten von Argumenten. Solche, die in einem Speicherplatz unterzubringen sind und andere, die zwei Byte brauchen. Mit dem Befehlswort (hier also LDA) zusammen, existieren in Assembler also 1-Byte-Befehle, 2-Byte-Befehle und 3-Byte-Befehle.

PC	IRQ	NV-BDIZC	AC	XR	YR	SP
E147	EA31	10110000	00	00	00	F8

Bild 5. Eine Registeranzeige

N	V	—	B	D	J	Z	C
Negativ-Flagge	Überlauf-Flagge	unbenutzt	Abbruch-Flagge	Dezimal-Flagge	Interrupt-Flagge	Zero-(Null) Flagge	Carry-(Übertrag) Flagge

Bild 6. Das Prozessor-Status-Register P: die Flaggen

Befehlswort	Adressierung	Byteanzahl	Code HEX	Code DEZ	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	unmittelbar	2	A9	169	2	N, Z
	absolut	3	AD	173	4	N, Z
LDX	unmittelbar	2	A2	162	2	N, Z
	absolut	3	AE	174	4	N, Z
LDY	unmittelbar	2	A0	160	2	N, Z
	absolut	3	AC	172	4	N, Z
STA	absolut	3	8D	141	4	keine
STX	absolut	3	8E	142	4	keine
STY	absolut	3	8C	140	4	keine
RTS	implizit	1	60	96	6	keine

Bild 7. Die ersten sieben Befehle

Das Argument von LDA ist also das, was in den Akku soll. Laden wir deshalb mal eine 1 in den Akku:

1500 LDA #$01

Wir haben jetzt einen 2-Byte-Befehl erzeugt. Was aber bedeuten »#« und »$« dabei? $ ist leicht zu erklären. Die große Mehrzahl der Assembler nimmt bei Zahlenangaben Hexadezimalzahlen an. Bei einigen muß man dies durch das $-Zeichen kennzeichnen. Manche Assembler lassen auch Binärzahlen, Dezimalzahlen und sogar ASCII-Zeichen als Argumente zu. Für jede Eingabeart steht dann vor dem Argument ein Zeichen, das die Art des Argumentes angibt, zum Beispiel häufig »!« für Dezimalzahlen oder »%« für Binärzahlen. Nun zum #-Zeichen. Es gibt viele Arten, den Akku zu laden. Direkt mit einer Zahl – wie wir hier —, aber zum Beispiel auch mit dem Inhalt eines anderen Speichers und so weiter. Man spricht von der sogenannten Adressierung.

Es gibt eine ganze Menge davon und jede wird auf eindeutige Weise gekennzeichnet. Wenn wir in unserem Akku eine Zahl laden, dann ist das die »unmittelbare« Adressierung und die kennzeichnet man mit dem #-Zeichen.

Wenn in Speicherstelle $1500 die Codezahl für LDA steht, dann muß die 1 in der Speicherstelle $1501 stehen, wie es sich für einen 2-Byte-Befehl gehört. Wenn Sie nun die Assemblerzeile eingegeben haben und ⟨RETURN⟩ drücken, dann taucht auf dem Bildschirm eine Fehlermeldung auf (bei vielen Assemblern). Wir müssen vorher nämlich noch unserem Software-Instrument sagen, jetzt zu assemblieren. Wie das geschieht, ist auch wieder von Assembler zu Assembler verschieden. Die meisten erwarten, daß man vor der Zeile noch ein A eingibt (zum Beispiel bei dem C 128):

A 1500 LDA #$01

Wenn Sie jetzt ⟨RETURN⟩ drücken, zeigt der Bildschirm:

A 1500 LDA #$01
A 1502

und meistens einen blinkenden Cursor, der auf die nächste Eingabe wartet. $ 1502 ist die nächste freie Speicherstelle, und wenn beim Programmablauf der Programmzähler nach dem LDA #$01 auf $1502 deutet, dann erwartet er dort den nächsten Befehl. Wenn dort Unsinn steht, dann stürzt der Computer im allgemeinen ab, je nachdem, welcher Code dann hier zufällig enthalten ist. Wir haben ja 256 Möglichkeiten dafür: $00 bis $FF. Im Gegensatz zu Basic, wo man durch den Interpreter die Möglichkeit hat, Zeilennummern zu bauen wie man will, muß hier das Programm eine ununterbrochene Perlenschnur von Befehlen in Speicherstellen sein. Durch einige Befehle läßt sich dieses Prinzip allerdings durchbrechen.

Damit wir die Wirkung von Befehlen sehen können, greife ich auf einen Befehl vor, der ähnlich dem STOP in Basic einen Programmabbruch bewirkt: BRK. Die genaue Funktion soll erst später erklärt werden, aber wir sehen jedenfalls dann, wenn ein Maschinenprogramm auf einen BRK-Befehl läuft, die Registerinhalte angezeigt. Das ist in den meisten Assemblern eingebaut. Wir ergänzen jetzt:

A 1502 BRK

Damit erstmal genug. Steigen Sie aus dem Assembler aus und starten Sie das Programm. In den meisten Assemblern geht das mit

G 1500

oder sonst von Basic aus mit SYS 5376. Jetzt werden wieder die Register angezeigt. Der Programmzähler steht auf 1503, im Akku steht 01, alle Flaggen außer der Breakflagge sind Null (die unbenutzte Flagge steht immer auf 1). Jetzt ändern wir das Argument:

A 1500 LDA #$00
A 1502 BRK

Wir starten wieder und sehen uns die Register an: Programmzähler 1503, Akku jetzt 00, aber bei den Flaggen hat

sich etwas verändert: Die Zero-Flagge ist auf 1 gesetzt. Wir sehen also: Diese Flagge bleibt so lange ungesetzt, solange nicht eine Null im Akku auftaucht, erst dann wird sie 1.
Noch einmal ändern wir das Programm:

```
A 1500 LDA #$FF
A 1502 BRK
```

Nach erneutem Start steht das Erwartete in den Registern, nur bei den Flaggen ist etwas Merkwürdiges passiert: Die Vorzeichenflagge steht auf 1. Das bedeutet, im Akku soll eine negative Zahl stehen! Nun wissen wir aber, daß $FF = dez. 255 ist. Dieses Rätsel wird uns noch eine Weile begleiten. Es sei hier nur bemerkt, daß kein Fehler vorliegt: Immer wenn in einer Zahl das Bit 7 gleich 1 ist, geht die Vorzeichenflagge auf 1. Die Lösung des Rätsels werden wir bei den negativen Binärzahlen finden.

Wir schließen aus alledem: Der LDA-Befehl beeinflußt die Vorzeichen- und die Zeroflagge.

9. Die absolute Adressierung

STA heißt »STore Accumulator«, also »lege Akkuinhalt ab«. Wie Sie sich denken können, muß auch hier ein Argument auftauchen, nämlich wohin abgelegt werden soll. Wir legen unseren Akkuinhalt in die erste Bildschirmspeicherstelle (C 64:$0400, VC 20 Grundversion: $1E00, VC 20 mit Erweiterung: $1000). Unser Programm muß also so aussehen:

```
A 1500 LDA #$01
A 1502 STA $0400
oder die entsprechende Adresse
(siehe oben).
```

Mit diesem STA-Befehl lernen wir eine neue Adressierungsart kennen: die »absolute« Adressierung. Sie ist daran zu erkennen, daß kein besonderes Merkmal verwendet wird. Die Adresse $ 0400 ist nicht in einem Byte darstellbar, sondern wird aufgeteilt auf zwei Bytes. Im Speicher steht jetzt:

```
1500 LDA #
1501 $ 01
1502 STA
1503 $ 00        »das ist das LSB«
1504 $ 04        »das ist das MSB«
```

Hier liegt also ein 3-Byte-Befehl vor und die nächste freie Speicherstelle ist $ 1505.

Vom Basic her wissen Sie, daß 1 der Bildschirmcode für den Buchstaben A ist und daß man jeder Bildschirmspeicherstelle auch eine Bildschirmfarbspeicherstelle zuordnet. Um ein eingeschriebenes Zeichen vom Hintergrund abzuheben, muß man dort dann eine Farbinformation eingeben. Der Start dieses Bildschirmspeichers liegt so:

```
C 64:              $ D800
VC 20 (Grundv.):   $9400
VC 20 (Erw. Vers.): $9600.
```

Der Farbe Schwarz entspricht die Codezahl 0. Wir ergänzen unser Programm durch:

```
A 1505 LDA #$00
```

A 1507 STA $D800 (oder entsprechender Speicher, siehe oben). Die nächste freie Adresse ist nun $150A. Unser Programm soll jetzt abgeschlossen sein. Damit der Computer aber beim Programmzählerstand $150A nicht Unsinn vorfindet, muß – ähnlich wie bei END in Basic – das Programm auf irgendeine Weise beendet werden. Das kann durch BRK geschehen. Wir wollen aber den dritten Assembler-Befehl kennenlernen:

```
RTS
```

Das heißt »Return From Subroutine«, also »Rückkehr aus Unterprogramm«. In unserem Fall bewirkt das eine Rückkehr zum Basic. Wie Sie sehen, ist das ein 1-Byte-Befehl, also ohne Argument. Auch hier spricht man von einer Adressierungsart, nämlich der »impliziten«-Adressierung. Man erkennt sie am Fehlen des Argumentes. Die Adresse ist implizit, das heißt im Befehl selbst enthalten. Dies ist nämlich ein Befehl, der immer an den Programmzähler gerichtet ist. Der Computer holt sich vom Stapel-Speicher die dort zuoberst liegende Adresse, das ist die, bei der der Computer in ein Unterprogramm gesprungen ist oder aber die, bei der der Computer Basic verlassen hat. Wir ergänzen also noch:

```
A 150A RTS
```

und starten das Programm, zum Beispiel von Basic aus mit SYS 5376. Natürlich taucht dann in der linken oberen Ecke des Bildschirmes ein schwarzes A auf. Hier noch der Basic-Lader:

```
10 FOR I=5376 TO 5386:READ A:POKE I,A:NEXT I:END
20 DATA 169,1,141,0,4*,169,0,141,0,216*,96.
```

Die mit * markierten Zahlen müssen für den VC 20 verändert werden: Grundversion: 30 und 148.
Erweiterung: 16 und 150.

10. Vier neue Befehle

Eine Kombination von LDA mit STA ist vergleichbar mit dem POKE-Befehl in Basic. Man kann in Assembler nicht direkt eine Zahl in einen Speicher einschreiben, sondern muß den Umweg über den Akku machen. Außer dem Akku eignen sich dazu aber auch das X-Register und das Y-Register. Hierfür gibt es die Befehle LDX (lade X-Register), STX (lege X-Register-Inhalt ab), LDY (lade Y-Register) und schließlich STY (lege Y-Register-Inhalt ab). Sie können das übungshalber an unserem kleinen Programm ausprobieren. An dem folgenden Programm sehen Sie noch eine Eigenart der drei Register (Akku, X-Register, Y-Register):

```
A 1500 LDA  #$01
A 1502 LDX  #$00
A 1504 LDY  #$02
A 1506 STA  $0400
A 1509 STX  $D800
A 150C STY  $0401
A 150F STX  $D801
A 1512 STA  $0402
A 1515 STX  $D802
A 1518 RTS
```

Für den VC 20 werden die entsprechenden Speicherstellen für Bildschirm- und Bildschirmfarbspeicher eingesetzt. Dieses Programm druckt – wie erwartet – »ABA« in die linke obere Ecke des Bildschirms. Dabei ist das X-Register dreimal ausgelesen worden und der Akku zweimal. Sie sehen also, daß die Registerinhalte durch die STA-, STX-, STY-Befehle nicht verändert werden.

Wir wollen noch etwas ausprobieren. Bisher haben wir den LDA-Befehl nur mit der »unmittelbaren« Adressierung kennengelernt. LDA, LDX, LDY können auch »absolut« adressiert werden.

```
A 1518 LDA $D800
```

Damit laden wir den Inhalt der Speicherstelle $ D800 (beim VC 20 die anderen Adressen des Bildschirmfarbspeichers) in den Akku. Der Inhalt ist seit $1509 eine Null. Jetzt weiter:

```
A 151B STA $0403
A 151E STX $D803
A 1521 RTS
```

Das müßte beim Ablauf des Programms noch einen Klammeraffen (@mit Bildschirmcode 0) an die vierte Stelle plazieren, was Sie durch SYS 5376 leicht nachprüfen können. Sie sehen, daß man mit diesen sieben Befehlen schon eine Menge anfangen kann.

Wir kommen noch einmal zur Adressierung. Ich hatte Ihnen gesagt, daß LDA #$01 ein 2-Byte-Befehl mit unmittelbarer

Adressierung ist (ein Byte für LDA und eines für 01), LDA $D800 ist ein 3-Byte-Befehl (ein Byte für LDA, je eines für das LSB und das MSB von $D800) mit absoluter Adressierung. Da werden Sie sich doch sicher schon gefragt haben, wo die Adressierung bleibt! Wenn aber kein Byte für die Adressenmarkierung (zum Beispiel #) reserviert ist, muß die Kennzeichnung irgendwie anders sein. Wenn Sie einen Disassembler zur Verfügung haben, dann sehen Sie sich damit unser Programm an. Fast jeder Disassembler gibt neben dem Assemblertext auch Byte für Byte in Hexadezimalzahlen die Codes an. Wenn Sie nun die beiden Befehle LDA #$01 und LDA $d800 von den Codes her untersuchen, sehen Sie folgendes:

```
1500 A9 01      LDA#$01
```

und

```
1518 AD 00 D8  LDA $D800
```

Offensichtlich gehört jeweils das erste angezeigte Byte zu LDA. Sie sind aber verschieden! Wir sehen daraus, daß die Codezahl für einen Befehl gleich zwei Informationen enthält: das Befehlswort selbst (LDA) und die Adressierungsart.

Genauso wie man LDA sowohl unmittelbar als auch absolut ausführen kann, ist das auch mit LDX und LDY möglich. Bei den Befehlen STA, STX, STY ist eine unmittelbare Adressierung sinnlos. Für RTS kennt man nur eine implizite Adressierung. Wir fassen das alles in Bild 7 zusammen.

In den letzten Spalten von Bild 7 ist noch angegeben, inwieweit durch diese Befehle das Prozessorstatusregister beeinflußt wird, so wie wir es für den Befehl LDA schon ausprobiert haben. In der vorletzten Spalte sehen Sie, wie lange die Ausführung eines Befehls dauert. Wenn sie für einen Taktzyklus etwa eine Mikrosekunde rechnen, dann müßten Sie jetzt ausrechnen können, wie lange unser letztes Programm zur Bearbeitung braucht: 48 Mikrosekunden. Ein vergleichbares Basic-Programm braucht dazu etwa hundertmal so lange: zirka 0,05 Sekunden.

11. Die Zahlen der Assembler-Alchimisten

Ein bißchen von Assembler-Alchimie verstehen Sie jetzt schon mit diesen sieben Befehlen. Wir wollen uns nun die Zahlen ansehen, die hier Verwendung finden: das Binärsystem und das Hexadezimalsystem.

Die einzigen Ziffern, die unser Computer kennt, sind 0 und 1. Sie stehen für »Strom an« oder »Strom aus«, oder für »keine magnetische Erregung« oder »magnetische Erregung«. Deshalb ist es für uns als angehende Assembler-Alchimisten von großer Bedeutung – wir arbeiten ja ganz eng an der Hardware – dieses binäre Zahlensystem handhaben zu können. Das Hexadezimalsystem kennt der Computer eigentlich gar nicht. Wir verwenden es deswegen, weil es in einem besonders engen Zusammenhang mit Binärzahlen und dem Aufbau unseres Computers steht: Die größte einstellige Hex-Zahl ist $F, das entspricht genau 1111 im Binärsystem, also dem maximalen Füllungsgrad eines halben Bytes, das Nibble genannt wird. Ein ganzes Byte kann maximal $FF enthalten (binär 1111 1111) und der gesamte Speicheradressenbereich unseres Computers geht bis $FFFF (dezimal 65535). Eine einstellige Hex-Zahl paßt also in ein Nibble, eine zweistellige in ein Byte und eine dreistellige oder vierstellige in zwei Byte, weshalb man solche Hex-Adressen auch recht leicht in das LSB und das MSB (auch Low- und High-Byte genannt) aufteilen kann:

```
$  D8  00
   MSB LSB
```

Rechnen werden wir mit Hexadezimalzahlen nicht, dazu benutzen wir dann das Dezimalsystem oder – wenn es sich um computerinterne Vorgänge handelt – das Binärsystem.

Das Rechnen mit Binärzahlen funktioniert genauso wie das mit Dezimalzahlen. Es gilt also

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

wobei binär 10 gleich dezimal 2 ist. Als Beispiel können wir mal 2+1=3 im Binärsystem rechnen:

```
  10 entspricht dez. 2
+01 entspricht dez. 1
-------------------
  11, was ja dezimal 3 ergibt.
```

Die Addition erfolgt also spaltenweise wie beim gewohnten dezimalen Addieren. Auch mit dem Übertrag läuft es wie im dezimalen. Beispiel: 2+2=4:

```
  10 entspricht dez. 2
+10 entspricht dez. 2
-------------------
 100, was dezimal eine 4 ergibt.
```

In der zweiten Spalte wurde nach der Regel verfahren: 1+1=10. Rechnen wir noch 3+3=6:

```
  11 entspricht dez. 3
+11 entspricht dez. 3
-------------------
 110, was dezimal eine 6 ergibt.
```

In der ersten Spalte wurde gerechnet 1+1=10, wobei nach dem alten Motto: 0 hin, 1 im Sinn die 0 unter den Strich gesetzt wurde. In der zweiten Spalte wird dann so verfahren: 1+1+1 (das ist die 1, die wir »im Sinn« hatten)=11. Ich meine, daß Sie ohne Probleme die folgenden Übungsaufgaben lösen und dann jeweils dezimal das Ergebnis nachprüfen können: 10+5, 7+1, 16+16, 240+16, 62+65.

12. Eine Zauberformel der Assembler-Alchimisten: INX, INY, INC, DEX, DEY, DEC?

Wir wissen ja schon, daß man diese »Zauberformeln« entzaubern kann. INX heißt einfach »INCrement X-Register«, also Inhalt des X-Registers um 1 erhöhen. Es wird Ihnen sicher einleuchten, daß INY dasselbe mit dem Y-Register tut. Etwas weniger deutlich ist das bei INC. Das bedeutet »INCrement memory«, also zähle zum Inhalt einer Speicherstelle eins dazu. INX und INY enthalten alles, was dem Computer zu sagen ist, sind also offensichtlich 1-Byte-Befehle mit der in der letzten Folge schon kennengelernten impliziten Adressierung. Bei INC muß dem Computer noch gesagt werden, welche Speicherstelle er um 1 erhöhen soll. Es gehört also noch eine Adresse dazu. Das läßt diesen Befehl im allgemeinen zu einem 3-Byte-Befehl werden.

Das Umgekehrte leisten die Befehle DEX, DEY und DEC. Sie bedeuten nämlich »DECrement X-Register«, also »zähle das X-Register um eins herunter«, beziehungsweise das Y-Register oder – bei DEC – die angegebene Speicherstelle. Für die Adressierungsart und die Anzahl Bytes pro Befehl gilt hier das gleiche wie für die INX...-Befehle. Sehen wir uns das an einem kleinen Beispiel an:

```
1500    LDA #00
1502    LDX #01
1504    STA D800
1507    STX 0400
150A    INX
150B    STA D801
150E    STX 0401
1511    DEX
1512    STA D802
1515    STX 0402
1518    BRK
```

Wenn Sie das kleine Programm mit G 1500 starten, dann sollten Sie in der linken oberen Ecke des Bildschirms ABA in schwarzer Schrift stehen haben. Was ist geschehen? Wir haben den Inhalt des Akkus (=0, also Farbcode für schwarz) in das Bildschirm-Farbregister geschrieben (#D800), dann den Inhalt des X-Registers (1 = POKE-Code für den Buchstaben A) in die erste Bildschirm-Speicherzelle (#0400). Anschließend wurde das X-Register um 1 erhöht (2 = POKE-Code für den Buchstaben B) und dieser Inhalt in die zweite Bildschirmzelle geschrieben. Außerdem mußte natürlich auch dieser Bildschirm-Farbspeicherplatz mit dem Farbcode 0 belegt werden. Durch DEX wurde das X-Register wieder heruntergezählt, somit wieder ein A erzeugt und in die dritte Bildschirmstelle gedruckt.

Sie haben sicher schon bemerkt, daß man auf diese Weise Abläufe mitzählen kann. Soll zum Beispiel ein Vorgang 20 mal wiederholt werden, dann packt man ins X-Register (oder ins Y-Register oder in eine andere Speicherstelle) den Anfangswert 0, läßt den Computer eine Arbeit ausführen, erhöht das entsprechende Register oder die Speicherzelle um 1 mit INX, INY oder INC, prüft dann, ob dieser Inhalt schon 20 geworden ist und so weiter. Wie man diese Prüfung vornimmt, dazu kommen wir erst später bei den BRANCH-Befehlen. Das ist also ähnlich wie in Basic bei den FOR...NEXT-Schleifen: Dort wird eine Variable als Zähler verwendet, hier ein Register (oder eine Speicherstelle). Ebenso wie in Basic bei diesen Schleifen kann man auch hier rückwärts zählen mit DEX, DEY oder DEC. Das hat oft gewisse Vorzüge, was uns aber noch nicht kümmern soll.

Wenn wir diese Befehle als Zähler verwenden, sollten wir im Auge behalten, daß eine Speicherstelle (auch ein X- oder Y-Register) Zahlen nur von 0 bis 255 enthalten kann. Die höchste 8-Bit-Zahl ist ja:

```
dez. 255 = bin. 1111 1111
     +1               1
                ----------
ergibt:         (1) 0000 0000
```

Wenn wir also über 255 hinauszählen, ergibt sich wieder 0 und so weiter, weil ein Überlauf stattgefunden hat. Das 9.Bit paßt nicht mehr in das Byte hinein. Um nochmal genau sehen zu können, was unser Computer da tut, probieren Sie einmal aus:

```
1500 LDA #01
1502 BRK
```

Das soll uns die Register zunächst mal im Ausgangszustand zeigen. Nach G 1500 werden sie angezeigt:

```
AC  XR  YR  N  V - BD I ZC
01  00  00  0  0 1 1 0 0 0 0
```

Im Akku steht jetzt die dort eingeladene 1. Nun wollen wir das X-Register laden mit 255 (also $FF). Dazu ändern wir das Programm:

```
1502 LDX #FF
1504 BRK
```

Nach erneutem G 1500 zeigen die Register:

```
AC  XR  YR  N  V - BD I ZC
01  FF  00  1  0 1 1 0 0 0 0
```

Im X-Register steht nun die Zahl $FF. Bei den Flaggen hat sich die N-Flagge (die negative Zahlen anzeigen soll) auf 1 geschaltet!

Nun wollen wir das X-Register über 255 hinauszählen. Wir verändern das Programm nochmal:

```
1504 INX
1505 BRK
```

Der Start mit G 1500 liefert uns die folgende Registeranzeige:

```
AC  XR  YR  N  V - BD I ZC
01  00  00  0  0 1 1 0 0 1 0
```

Wie erwartet, ist der Überlauf des X-Registers eingetreten: Es ist jetzt Null. Die N-Flagge hat ihren gewohnten Wert 0 wieder angenommen und die Z-Flagge, die uns anzeigt, ob die letzte Operation eine Null erzeugt hat, ist jetzt gesetzt. Bei weiterem Hochzählen verschwindet die Z-Flagge wieder:

```
1505 INX
1506 BRK
```

G 1500 liefert den Registerinhalt:

```
AC  XR  YR  N  V - BD I ZC
01  01  00  0  0 1 1 0 0 0 0
```

Das gleiche passiert bei Verwendung des Y-Registers als Zähler, wie Sie leicht durch Austauschen aller auf X bezogenen Befehle feststellen können. Sehr nett ist es, diesen Befehlsablauf einmal für den INC-Befehl auf die Speicherstelle $0400 (Bildschirmspeicher links oben) bezogen ablaufen zu lassen. Wenn man darauf achtet, daß kein Hochscrollen des Bildschirms eintritt, kann man das Ergebnis außer in den Registern auch noch als Zeichen auf dem Bildschirm verfolgen. Der Beginn der Befehlsequenz ist dann sinnvollerweise:

```
1500 LDA #FF
1502 STA 0400
1505 BRK
```

Im folgenden setzt man dann anstelle von INX immer INC 0400 ein.

Was passiert beim Herunterzählen unter Null? Sie können das mit der gezeigten Befehlskette leicht verfolgen, indem Sie immer statt INX jetzt DEX setzen und die Register nicht mit $FF, sondern mit 01 laden. Es zeigt sich, daß beim Herabzählen nach der Null wieder 255 (=$FF) im Register zu finden ist. Die Reaktion der N- und der Z-Flagge auf den jeweiligen Registerinhalt ist die gleiche wie beim Hochzählen.

Es ist uns nun deutlich, daß diese sechs Befehle die N-Flagge und die Z-Flagge beeinflussen können. Diese Tatsache wird später noch eine große Rolle spielen, wenn es um die bereits erwähnte Schleifenkontrolle geht.

13. Noch ein alchimistischer Zahlentrick: BCD

Die Assembler-Alchimisten haben noch viel mehr Arten der Zahlen- und Zeichendarstellung auf Lager. Eine davon ist die Codierung als BCD-Zahlen. BCD kommt vom englischen »binary coded dezimal«, was bedeutet: Binär codierte Dezimalzahlen.

Zwischendurch möchte ich noch eine Bemerkung loswerden, die Sie als Trost auffassen sollen: Auch wenn wir später andere Zahlendarstellungen kennenlernen werden, es wird nicht so schwierig! Sogar so komplette Idioten wie Computer verstehen das, obwohl man ihnen alles haarklein vorkauen muß.

Wenden wir uns nun wieder den einfachen BCD-Zahlen zu. Alle Zahlen von 0 bis 9 lassen sich binär mit nur 4 Bit ausdrücken:

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

Die weiteren Werte 1010 bis 1111 werden in der BCD-Codierung nicht benutzt. Liegt nun eine Dezimalzahl (zum Beispiel 12) vor, dann wird jede Stelle dieser Zahl (also die 1

und die 2) getrennt binär codiert. In unserem Beispiel mit der 12 wäre das dann 0001 für die 1 und 0010 für die 2. Somit ist die 12 im BCD-Code 0001 0010. Jede Ziffer erhält so ihr Nibble. Eine Zahl im BCD-Format hat deswegen keine feste Anzahl von Bytes, sondern die Byte-Zahl hängt von der Anzahl der Stellen ab. Die Zahl 1984 beispielsweise braucht 2 Byte: 0001 1001 1000 0100.

Schwierig gestaltet sich das Rechnen mit diesen Zahlen wegen der sechs unbenutzten Codes. Aber auch da habe ich einen Trost für Sie: Wir werden damit nicht rechnen. Wozu das ganze dann, werden Sie sich fragen? Der Grund für das alles ist, daß BCD-Zahlen im Gegensatz zu den Zahlen mit festem Format (die sonst verwendet werden) so eingegeben und verarbeitet werden können, wie sie vorliegen. Das ist im kaufmännischen Bereich manchmal notwendig, wo eben 1000 mal 0,1 Pfennige 1 Mark ergeben und Fehler unzulässig sind. Sollten Sie also vor dem Problem stehen, mit BCD-Zahlen rechnen zu müssen, grämen Sie sich nicht: Unser Prozessor kennt den Dezimalmodus. Er ist dann eingeschaltet, wenn die Dezimal-Flagge auf 1 gesetzt ist.

Damit sollen Sie dann auch noch gleich zwei neue Befehle kennenlernen: SED und CLD. Der erstere hat nichts mit Parteien zu tun, sondern ist die Abkürzung für »SEt Dezimalflag«, also setze die Dezimalflagge. So schalten Sie den Dezimal-Modus ein. Wie Sie sicher schon messerscharf geschlossen haben, heißt CLD »CLear Dezimal-flag«, also setze die Dezimalflagge auf Null, wodurch dieser Modus wieder auszuschalten ist.

Wichtig! Wenn Sie argwöhnen, daß in einem Programm irgendwann mal die Dezimal-Flagge gesetzt sein könnte, dann gehen Sie auf Nummer sicher und schieben Sie vor eine Rechenoperation, die nicht im Dezimalmodus laufen soll, ein CLD.

Beide Befehle sind 1-Byte-Befehle mit implizierter Adressierung. Sie beeinflussen lediglich die Dezimalflagge.

Wie schon mal betont: Der Computer ist strohdumm. Er kann nicht einmal auf normale Weise voneinander abziehen! Deswegen geht er den komplizierten Weg: Er addiert eine negative Zahl. Nur: Wie sehen negative Binärzahlen aus? Wir werden diese Frage in drei Etappen beantworten.

a) Man könnte eine Flagge setzen, die 1 ist bei negativen und 0 bei positiven Zahlen. Bei einigen Fließkommazahlen wird das auch so gemacht. Hier aber setzt man die Flagge direkt in die Zahl ein: Bit 7 jeder Zahl ist jetzt ein Vorzeichenmerkmal. Wenn dieses Bit 0 ist, handelt es sich um eine positive, wenn es 1 ist, um eine negative Zahl. Auf diese Weise ist also +1 wie bisher 0000 0001, wohingegen —1 jetzt 1000 0001 hieße. Damit wird allerdings der Zahlenbereich, der durch ein Byte auszudrücken ist, verschoben. 255=binär 1111 1111 kann so nicht mehr verwendet werden. Die größte Zahl, die jetzt ausgedrückt werden kann, ist 0111 1111 = dezimal 127. Die kleinste Zahl ist dann 1111 1111 = —127. Probieren wir mal aus, wie sich damit rechnen läßt:

```
+10        0000 1010
—6         1000 0110
-------------------
ergibt     1001 0000 = —16,
```

was offensichtlich falsch ist, denn nach Adam Riese sollte +4 herauskommen. So kann man also nicht rechnen!

Man nennt diese Art der Zahlendarstellung übrigens »signed binary«-Format, also in Deutsch: markierte Binärzahlen.

b) Der nächste Schritt ist das sogenannte Einerkomplement. Dabei tritt für die positiven Zahlen keine Änderung ein. Die negativen entstehen aus den positiven durch Komplementbildung, das heißt jedes Bit der positiven Zahl wird in sein Gegenteil verkehrt, wie es das folgende Beispiel zeigen soll:

0000 1100 ist +12,
dann ist das Einerkomplement:
1111 0011 = —12.

Interessanterweise taucht hier auch wieder das Merkmal der »signed binary«-Zahlen auf: die 1 in Bit 7 bei negativen Zahlen. Beschränkt man sich auf den Zahlenbereich, der für die »signed binary«-Zahlen gültig war, dann hätten wir jetzt beide Darstellungsweisen miteinander vereint. Nun müssen wir natürlich noch feststellen, ob man so auch rechnen kann.

```
+8          0000 1000
—6          1111 1001
in Einerkomplementdarstellung
-------------------
ergibt    (1) 0000 0001
```

was 1 mit einem Übertrag ergäbe, jedenfalls nicht 2, wie es sich gehört. Also ist auch die Einerkomplementdarstellung noch nicht das Gelbe vom Ei.

c) Ich will Sie nicht länger auf die Folter spannen: Wenn man zum Einerkomplement einer Zahl noch 1 dazuzählt, erhält man das Zweierkomplement. Und genauso werden negative Zahlen in unserem Computer gehandhabt. Die positiven Zahlen bleiben unverändert. Von den negativen bildet man das Zweierkomplement wie zum Beispiel hier mit der Zahl —12:

```
12    0000 1100   normale Binärdarstellung
—12   1111 0011   Einerkomplement
+1    0000 0001   addieren
-------------------
—12   1111 0100   Zweierkomplement
```

Jetzt wollen wir auch diese Zahlenart ausgiebig testen: Wir rechnen nochmal 8-6:

```
+8   0000 1000
—6   1111 1010   das ist —6 in der
                 Zweierkomplementdarstellung.
-------------------
ergibt
    (1) 0000 0010
```

also 2 mit einem Übertrag, der ignoriert wird. Das Ergebnis ist richtig. Wenn bei einer solchen Rechnung eine negative Zahl herauskommt, ist sie nicht leicht zu erkennen. In solchen Fällen kehrt man das Vorzeichen um, indem man das Zweierkomplement berechnet. Das machen wir mal am Beispiel 5—6:

```
+5   0000 0101
—6   1111 1010   das ist wieder unser Zweier-
                 komplement von 6, also —6
-------------------
ergibt1111 1111
```

das ist —1 in der Zweierkomplementdarstellung. Zur Kontrolle nun die Vorzeichenumkehr durch Umrechnen ins Zweierkomplement:

```
Einerkomplement davon   0000 0000
plus 1                  0000 0001
-------------------------------
ergibt                  0000 0001
```

also wie erwartet +1.

Auf diese Weise rechnet unser Computer mit negativen Zahlen. Negative ganze Zahlen speichert er im Zweierkomplement-Format. Auch wenn wir nun etwas vorgreifen müssen, wollen wir uns das ansehen. Dazu schalten Sie am besten erst einmal den Computer aus und laden dann den SMON beziehungsweise ihren Assembler. Dann bauen wir ein kleines Basic-Programm:

```
10 A%=—12
20 END
```

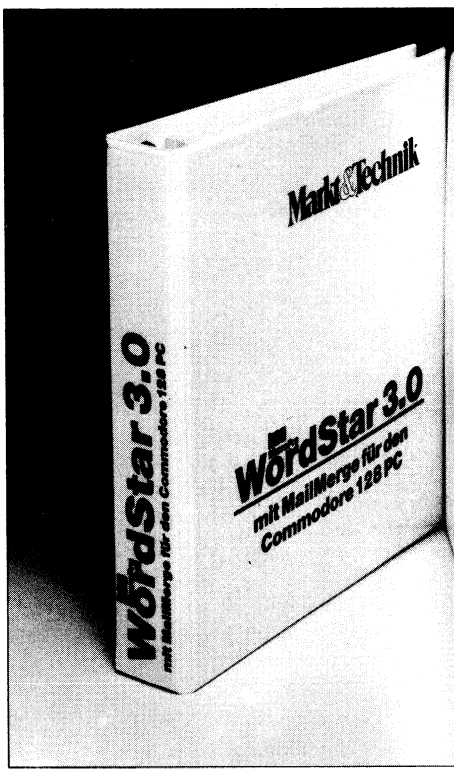
14. Wie Variable im Speicher stehen

Noch nicht RUN eingeben! Zuerst schalten Sie den Maschinensprachmonitor ein und wir sehen uns das Programm so an, wie es im Speicher steht. Der Basic-Speicher des C 64

MicroPro®

ASHTON·TATE

WordStar, dBASE



Der Bestseller unter den Textverarbeitungsprogrammen für PCs bietet Ihnen bildschirmorientierte Formatierung, deutschen Zeichensatz und DIN-Tastatur sowie integrierte Hilfstexte. Mit MailMerge können Sie Serienbriefe mit persönlicher Anrede an eine beliebige Anzahl von Adressen schreiben und auch die Adreßaufkleber drucken.

WordStar/MailMerge für den Commodore 128 PC
Bestell-Nr. MS 103 (5¼"-Diskette)
Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle. **DM 199,-* (sFr. 178,-)**

---

dBASE II, das meistverkaufte Programm unter den Datenbanksystemen, eröffnet Ihnen optimale Möglichkeiten der Daten- u. Dateihandhabung. Einfach u. schnell können Datenstrukturen definiert, benutzt und geändert werden. Der Datenzugriff erfolgt sequentiell oder nach frei wählbaren Kriterien, die integrierte Kommandosprache ermöglicht den Aufbau kompletter Anwendungen wie Finanzbuchhaltung, Lagerverwaltung, Betriebsabrechnung usw.

dBASE II für den Commodore 128 PC
Bestell-Nr. MS 303 (5¼"-Diskette)
Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle. **DM 199,-* (sFr. 178,-)**

Wenn Sie die zeitraubende manuelle Verwaltung tabellarischer Aufstellungen mit Bleistift, Radiergummi und Rechenmaschine satt haben, dann ist MULTIPLAN, das System zur Bearbeitung »elektronischer Datenblätter« genau das richtige für Sie! Das benutzerfreundliche und leistungsfähige Tabellenkalkulationsprogramm kann bei allen Analyse- und Planungsberechnungen eingesetzt werden, wie z.B. Budgetplanungen, Produktkalkulationen, Personalkosten usw. Spezielle Formatierungs-, Aufbereitungs- und Druckanweisungen ermöglichen außerdem optimal aufbereitete Präsentationsunterlagen!

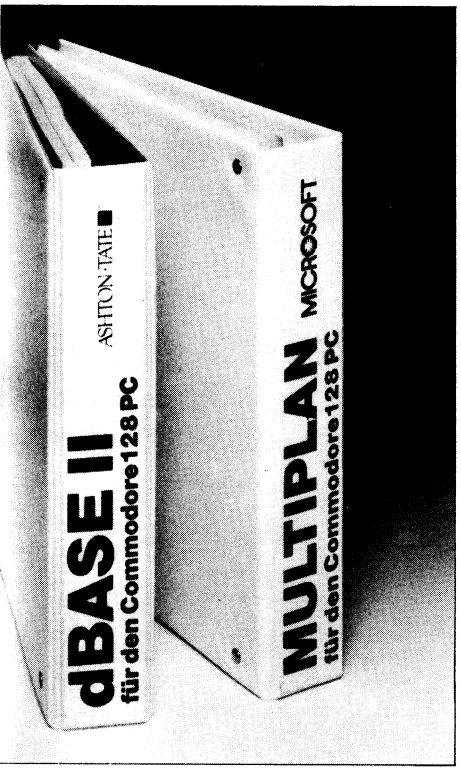
MULTIPLAN für den Commodore 128 PC
Bestell-Nr. MS 203 (5¼"-Diskette)
Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle. **DM 199,-* (sFr. 178,-)**

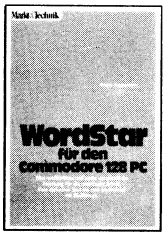
Dies sind die ersten drei weltbekannten Software-Produkte für den Commodore 128 PC. Weitere folgen in Kürze!

Diese Markt&Technik-Softwareprodukte erhalten Sie in den Computer-Abteilungen der Kaufhäuser Horten, Karstadt, Kaufhof, Quelle oder bei Ihrem Computerhändler.
Wenn Sie direkt beim Verlag bestellen wollen: per Nachnahme oder gegen Vorauskasse durch Verrechnungsscheck oder mit der eingehefteten Zahlkarte.
Bestellungen im Ausland bitte an nebenstehende Adressen.
Für Auskünfte stehen Ihnen Herr Barsa, Tel. 089/46 13-133, und Herr Teller, Tel. 089/46 13-205, gerne zur Verfügung.

MICROSOFT®

I und MULTIPLAN





Mit diesem Buch haben Sie eine wertvolle Ergänzung zum WordStar-Handbuch: Anhand vieler Beispiele steigen Sie mühelos in die Praxis der Textverarbeitung mit **WordStar** ein. Angefangen beim einfachen Brief bis hin zur umfangreichen Manuskripterstellung zeigt Ihnen dieses Buch auch, wie Sie mit Hilfe von MailMerge Serienbriefe an eine beliebige Anzahl von Adressen mit persönlicher Anrede senden können.
WordStar für den Commodore 128 PC
Best.-Nr. MT 780, ISBN 3-89090-181-6

Zu einem Weltbestseller unter den Datenbanksystemen gehört auch ein klassisches Einführungs- und Nachschlagewerk! Dieses Buch von dem deutschen Erfolgsautor Dr. Peter Albrecht begleitet Sie mit nützlichen Hinweisen, die nur von einem Profi stammen können, bei Ihrer täglichen Arbeit mit **dBASE II**. Schon nach Beherrschung weniger Befehle ist der Einsteiger in der Lage, Dateien zu erstellen, mit Informationen zu laden und auszuwerten.
dBASE II für den Commodore 128 PC
Best.-Nr. MT 838, ISBN 3-89090-189-1

Dank seiner Menütechnik ist **MULTIPLAN** sehr schnell erlernbar. Mit diesem Buch von Dr. Peter Albrecht werden Sie Ihre Tabellenkalkulation ohne Probleme in den Griff bekommen. Als Nachschlagewerk leistet es auch dem Profi nützliche Dienste.
MULTIPLAN für den Commodore 128 PC
Best.-Nr. MT 836, ISBN 3-89090-187-5

Jedes Buch kostet DM 49,- (sFr. 45,10).
Erhältlich bei Ihrem Buchhändler.

Sie erhalten jedes **WordStar-, dBASE II-** und **MULTIPLAN-Programm** für Ihren Commodore 128 PC fertig angepaßt (Bildschirmsteuerung und Druckerinstallation).

Jedes Programmpaket enthält außerdem ein ausführliches Handbuch mit kompakter Befehlsübersicht.

*** inkl. MwSt.**
Unverbindliche Preisempfehlung

Markt Technik
BUCHVERLAG

Hans-Pinsel-Straße 2, 8013 Haar bei München
Schweiz: Markt & Technik-Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, ☎ 042/41 56 56
Österreich: Microcomput-ique Schiller, Fasangasse 21, A-1030 Wien, ☎ 0222/78 56 61

beginnt im Normalfall bei $0800. Wir geben also den Monitorbefehl M 0800.

Uns genügen schon die Speicherplätze bis $081C. Nun sehen wir das nackte Basic-Programm im Speicher, so wie es uns C. Sauer in seinem Artikel »Der gläserne VC 20, Teil 1« im 64'er, Ausgabe 9/84 auf Seite 156 beschrieben hat.

In Bild 8 ist unser Speicherinhalt kommentiert zu sehen. Das Programm endet im Speicherplatz $0813. Das Kennzeichen für Programmende sind zwei aufeinanderfolgende Bytes mit dem Wort Null. Dahinter werden die Variablen abgelegt, sobald das Programm gestartet wird. Wir steigen aus dem Monitor durch X aus und starten das Programm mit RUN. Jetzt sehen wir nochmal in den Speicher. Bis $0813 hat sich nichts verändert. Danach aber ist jetzt in 7 Bytes die Variable A% abgelegt. Das zeigt Bild 9.

Zunächst einmal die Bytes $0814 und $0815: Hier wird der Variablenname und -typ angegeben. Der Typ ist aus den Bits 7 zu erkennen. Sind beide (wie hier) gleich 1, dann handelt es sich um eine Integervariable (also eine ganze Zahl). Läßt man die Kennbits außer acht, zeigt sich, daß in $0814 der Code für den Buchstaben A steht und $0815 nur den Wert 0 enthält. Nun zum Rest: Der C 64 legt Integers in nur 2 Byte ab - die restlichen 3 Byte $0818 bis $081A bleiben unbenutzt. Das ist auch dann der Fall, wenn danach noch weitere Variable kommen. Es bringt also keine Speicherersparnis (VC 20-Benutzer aufgepaßt!), wenn man mit Ganzzahlvariablen arbeitet!

In $0817 steht $F4, welches binär ausgedrückt 1111 0100 ist. Das kennen wir noch von weiter oben als die —12 im Zweierkomplement-Format. Woher kommt $FF in Speicherzelle $0816? Wie gesagt, die Integers werden in 2 Byte gespeichert, und wenn wir —12 in 16 Bit ausdrücken, dann sieht das so aus:

+12	0000 0000 0000 1100
Einerkomplement:	1111 1111 1111 0011
plus 1	0000 0000 0000 0001
ergibt —12:	1111 1111 (MSB =$FF) 1111 0100 (LSB =$F4)

als 16-Bit-Zweierkomplement.

Die größte positive ganze Zahl, die man in 2 Byte ausdrücken kann, ist 32767, was binär

0111 1111 1111 1111

ergibt. Die kleinste ist

1000 0000 0000 0000

also —32768. Das ist der Grund dafür, daß der C 64 Integers größer als 32767 oder kleiner als —32767 dankend mit ILLEGAL QUANTITY ERROR ablehnt, wenn sie als Argument verwendet werden. (Die Zahl —32768 kann als Ergebnis von logischen Operationen auftauchen.)

Damit will ich Sie erstmal von den Zahlenspielereien erlösen. Sie können die Art des Abziehens von Zahlen durch Addieren des Zweierkomplementes bis zum nächsten Mal an weiteren Beispielen üben. Wenn Sie das mit 16-Bit-Zahlen tun, werden Sie bald feststellen, daß noch nicht alles so funktioniert wie es sollte...

Wir können jetzt übrigens auch das Rätsel lösen, weshalb bei positiven Zahlen (zum Beispiel LDA #FF) die Negativ-Flagge auf 1 geht: Die Flagge wird immer dann gezückt, wenn eine Zahl auftritt, die in Bit 7 eine 1 aufweist. Ganz einfach, gell?

15. Ein wirkungsvolles Zweiglein: BNE

Vermutlich raucht Ihnen nach soviel Zahlensalat der Kopf. Deshalb sollen Sie zur Entspannung noch einen neuen Assembler-Befehl kennenlernen und auch gleich ein nützliches Programmbeispiel dazu.

BNE heißt »Branch if Not Equal zero«, was man übersetzen kann mit »verzweige, wenn ungleich Null«. Genauer gesagt: Es wird dann verzweigt - also zu einer angegebenen Adresse gesprungen —, wenn die Z-Flagge (die haben wir bei den INX,DEX...-Befehlen genauer untersucht) nicht gesetzt ist, also 0 zeigt. Sehen wir uns das mal an der nachfolgenden Verzögerungsschleife an, deren Flußdiagramm Bild 10 zeigt.

Das Programm dazu:

```
1500    LDX #FF
1502    LDY #FF
1504    DEY
1505    BNE 1504
1507    DEX
1508    BNE 1502
150A    BRK
```

Zunächst einmal werden das X- und das Y-Register als Zähler initialisiert (also mit einem Ausgangswert geladen). Mit dem vorhin behandelten Befehl DEY wird dann das Y-Register um 1 heruntergezählt, was jetzt $FE ergibt. Für die Nullflagge (Z) bedeutet das den Inhalt 0, denn es liegt kein Grund vor, sie zu setzen (also eine 1 dort anzuzeigen), weil noch keine Null aufgetreten ist. Bei der nachfolgenden Prüfung durch BNE wird also eine Verzweigung nach 1504 das Ergebnis sein, worauf das Y-Register weiter verringert und dann die Z-Flagge erneut geprüft wird und so weiter. Das geht so lange, bis nun wirklich endlich die Null im Y-Register erreicht ist. In diesem Fall zählt DEX nun das X-Register herunter und der nächste BNE-Befehl führt zum Sprung nach 1502, wo das Y-Register wieder auf $FF gesetzt wird. Auf diese Weise wird die äußere Schleife 255mal und die innere 65025mal durchlaufen.

0800	**00**	**0C**	**08**	**0A**	**00**	**41**	**25**	**B2**
		080C Koppeladresse		000A Zeilennr.10		A	%	= Token
0808	**AB**	**31**	**32**	**00**	**12**	**08**	**14**	**00**
	— Token	1	2	Zeilen-ende	0812 Koppeladresse		0014 Zeilennr.20	
0810	**80**	**00**	**00**	**00**	**FF**	**FF**	**FF**	**FF**
	END Token	Zeilen-ende	Programm-ende		Leerer Speicher			

Bild 8. Der Monitor zeigt das nackte Programm im Speicher

Speicherstelle	0814	0815	0816	0817	0818 bis 081A
Byte	1	2	3	4	5 — 7
Inhalt	C1 1100 0001	80 1000 0000	FF 1111 1111	F4 1111 0100	00 00 00 unbenutzt bei Integerzahlen
	Kennbits 7 für Integer 0100 0001 ≙ 65 Code für A		MSB von —12	LSB	
	Variablenname und -typ		Variablenwert		

Bild 9. So werden Integer-Variable aus Basic-Programmen vom C 64 im Speicher eingerichtet

Sie haben beim Eingeben des Programmes vermutlich etwas gestutzt, als der Assembler nach dem BNE 1504 als nächste Adresse statt dem erwarteten 1508 eine 1507 ausgegeben hat. Der Befehl sieht zwar wie ein 3-Byte-Befehl aus, ist aber nur ein 2-Byte-Befehl! Das liegt an der speziellen Art der Adressierung von solchen Branch-Anweisungen: Der sogenannten relativen Adressierung, die wir aber erst später mit den anderen Branch-Befehlen behandeln werden.

Wenn Sie das Programm mit G 1500 starten, werden Sie – obwohl alles in Maschinensprache schnell läuft – eine merkliche Verzögerung feststellen, bevor die Registeranzeige auftaucht. Noch längere Verzögerungen lassen sich ohne weiteres erreichen, indem man mehr Schleifen ineinanderschachtelt. Dabei verwendet man dann den DEC-Befehl.

In der Tabelle 2 sind auch die Zyklen angegeben, die die neu gelernten Befehle zur Abarbeitung benötigen. Mit solchen Angaben lassen sich recht genau definierte Zeiten einstellen, in denen der Computer nichts anderes tut als durch das Programm zu flitzen. Wozu das dient, braucht wohl kaum noch gesagt werden: Wenn Sie zum Beispiel einen Text auf dem Bildschirm lesen wollen, bevor das Programm weiterläuft oder wenn Sie mit Peripherie arbeiten, die langsamer als das Programm ist oder... Allerdings muß noch gesagt werden, daß es noch elegantere Methoden zur Verzögerungs-Programmierung gibt als das Lahmlegen des Computers, aber dazu kommen wir erst später.

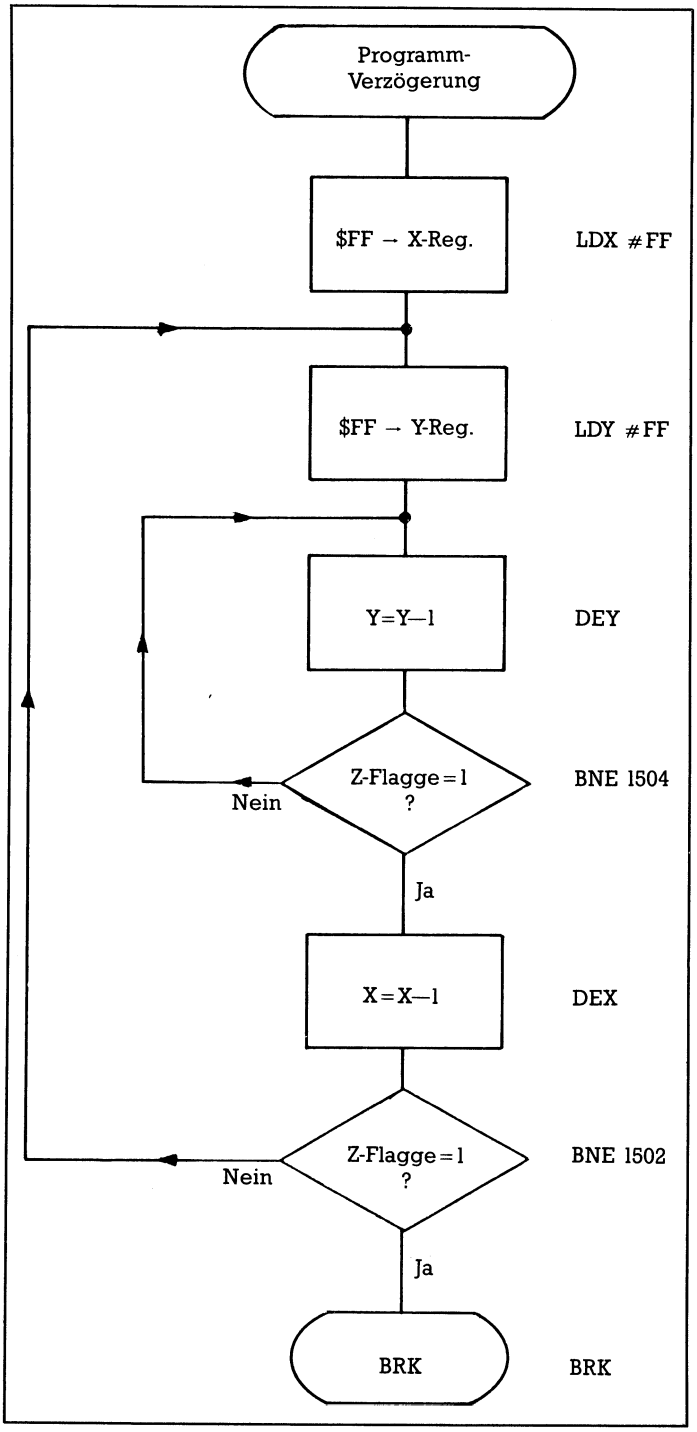


Bild 10. Flußdiagramm zur Verzögerungsschleife

16. Herr Carry und der V-Mann

Neun neue Befehle haben wir bisher kennengelernt und wir wissen nun, wie unser Computer ganze Zahlen (sogenannte Integers) abspeichert. Zur Erinnerung: Das geschieht im Zweierkomplement-Format. Das Bit 7 einer 8-Bit-Zahl dient dabei als Vorzeichen-Merkmal: Wenn es 0 ist, liegt eine positive Zahl vor, die genauso aussieht, wie wir bislang immer Binärzahlen kannten. Ist das Bit 7 aber eine 1, dann haben wir es mit einer negativen Zahl in der Zweierkomplement-Darstellung zu tun. Wenn wir – wie unser Computer – zur Verarbeitung ganzer Zahlen 16 Bits (also 2 Bytes) verwenden, dann ist eben Bit 15 anstelle von Bit 7 das Vorzeichenbit.

Wenn Sie ein bißchen mit solchen Zahlen gerechnet haben, konnten Sie sicher feststellen, daß zwar oft das richtige Ergebnis herauskam – aber leider nicht immer.

Keine Angst, wir sind nicht ins Krimi- oder Agentenmilieu gewechselt! Wir haben es mit zwei Flaggen zu tun, der Carry- und der V-Flagge. »To carry« heißt auf deutsch etwa »tragen«. In der Registeranzeige ist diese Flagge immer mit C gekennzeichnet. Was wird denn hier getragen? Das ergründen wir am besten an einem Beispiel. Dazu rechnen wir mit normalen Binärzahlen (also ohne Rücksicht auf Vorzeichenbits). Wir zählen die Zahlen 128 und 130 zusammen:

```
  128        10000000
+ 130      + 10000010
-----------------------
  258      (1)00000010
```

Das Ergebnis 258 ist richtig – auch in der Binärdarstellung – nur es paßt nicht mehr in 8 Bits. Ein Bit wurde überTRAGEN in ein extra dafür vorgesehenes Plätzchen: In das Carry-Bit. Jedesmal also, wenn so ein Übertrag in einer Rechenoperation des C 64 stattfindet, zeigt die Carry-Flagge eine 1 (Bild 11).

Je nach Art der von uns programmierten Aufgabe können wir nun dieses Carry-Bit weiterverarbeiten. Es gibt Situationen, in denen man es einfach ignorieren darf (dazu kommen

Befehls-wort	Adressierung	Byte-anzahl	Code Hex	Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
INX	implizit	1	E8	232	2	N,Z
INY	implizit	1	C8	200	2	N,Z
INC	absolut	3	EE	238	6	N,Z
DEX	implizit	1	CA	202	2	N,Z
DEY	implizit	1	88	136	2	N,Z
DEC	absolut	3	CE	206	6	N,Z
SED	implizit	1	F8	248	2	1 → D
CLD	implizit	1	D8	216	2	0 → D
BNE	relativ	2	D0	208	2	—
					+1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	

Tabelle 2. Die neuen Befehle

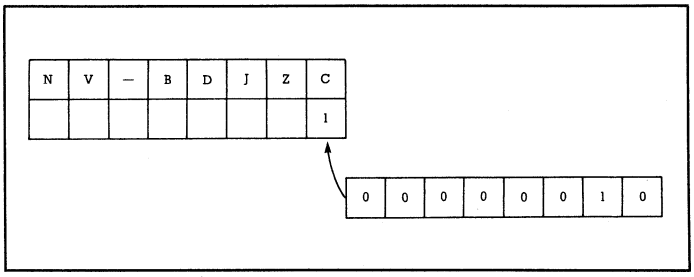


Bild 11. Das Carry-Bit als Bit 8 einer Rechenoperation

wir gleich noch) oder aber solche, wo man es weiter in der Rechnung verwendet. Schließlich kann es auch noch einen Fehler anzeigen: Dann nämlich, wenn das größte zulässige Ergebnis 1111 1111 sein darf. Natürlich kann das Carry-Bit auch gesetzt werden, wenn man in der Zweierkomplementform rechnet. Die Verhältnisse sind dann aber für ein leicht überschaubares Beispiel des Übertrages zu verwickelt, wie Sie gleich sehen werden.

Wenn wir nämlich mit den Zweierkomplement-Zahlen rechnen, dann interessieren uns auch Fälle wie bei der Addition von 64 und 66:

```
    64      01000000
+   66    + 01000010
-------------------
 (-126)     10000010
```

Das ist offensichtlich falsch. Bei der Addition ist durch das Zusammenzählen der Bits 6 plötzlich Bit 7 gesetzt worden. Da wir es aber mit einer Zweierkomplementzahl zu tun haben, bei der dieses Bit 7 eine negative Zahl anzeigt, folgt ein Fehler. Es ist also von Bedeutung, so einen Überlauf (englisch: 'overflow') erkennen zu können um eine entsprechende programmtechnische Reaktion zu starten. Es wird die Überlauf-Flagge V auf 1 gesetzt. Leider ist die Sache aber nicht so einfach, daß sie immer gesetzt würde, wenn von Bit 6 nach Bit 7 ein Übertrag stattfindet. Gesetzt wird diese V-Flagge nur in folgenden zwei Fällen:

1) Es findet ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag (wie beim Carry)

2) Es findet kein interner Übertrag von Bit 6 nach Bit 7 statt, aber ein äußerer Übertrag passiert.

Merken kann man sich das am besten so: Immer dann, wenn gewissermaßen das Vorzeichenbit 7 »versehentlich« verändert wurde, wird die V-Flagge auf 1 gesetzt. Das ist ein harter Brocken! Wir sind es ja gewohnt, daß wir uns um diese Dinge beim Computer eigentlich gar nicht mehr kümmern müssen. Außerdem würde das ja erfordern daß man sich bei allen Operationen vorher überlegen muß, welche Fehler also durch »versehentliches« Vorzeichenändern passieren können! Genauso ist es – in der Programmierpraxis wird Ihnen aber das ganze Problem nicht mehr so groß vorkommen. Wir wollen uns dieses Zusammenspiel der Überträge von Bit 6 nach Bit 7 und von Bit 7 nach Bit 8 (also in Carry-Bit) noch anhand einiger Beispiele klarer machen.

Im obigen Beispiel der Addition von 64 und 66 haben wir einen Fall schon behandelt: Es fand ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag in Carry-Bit. Deswegen wurde dann auch die V-Flagge gesetzt. Das Problem läßt sich hier ganz einfach lösen zum Beispiel durch Verwendung von 16-Bit-Zahlen:

```
    64      0000000001000000
+   66    + 0000000001000010
-------------------------
   130      0000000010000010
```

Bei 16-Bit-Zahlen ist ja Bit 15 das Vorzeichenbit, welches hier keine Änderung erfährt.

Der andere Fall tritt auf bei der Addition von zwei negativen Zahlen wie -125 und -64:

```
  -125      10000011
-  64      11000000
-------------------
 (+67)   (1)01000011
```

Auch das ist offensichtlich falsch: Es hat wieder »versehentlich« ein Vorzeichenwechsel stattgefunden. Dies ist also der Fall, wo zwar ein Übertrag ins Carry-Bit stattfand aber kein Übertrag von Bit 6 nach Bit 7. Auch dieses Problem läßt sich durch Verwendung von 16-Bit-Zahlen lösen. Eine kleine Trainingsaufgabe für Sie!

Man kann also sagen: Immer dann, wenn bei 8-Bit-Rechnungen der mittels Zweierkomplementzahlen darstellbare Bereich (127 bis -128) über- oder unterschritten wird, fuhrwerkt man im Vorzeichen-Bit herum und verfälscht das Ergebnis. Dann leuchtet wie eine rote Ampel die Überlauf(V)-Flagge auf und sagt uns, daß wir besser in diesen Fällen mit 16-Bit-Zahlen arbeiten sollten.

Nun noch zum Ignorieren des Carry-Bits, das ich weiter oben erwähnt habe. Bei allen 8-Bit-Rechenoperationen mit Zweierkomplementzahlen kann das Carry-Bit vernachlässigt werden. Zwei Beispiele sollen das wieder illustrieren. Wir addieren +4 und -2:

```
    +4      00000100
+   -2    + 11111110
-------------------
    +2    (1)00000010
```

Das Carry-Bit wird außer acht gelassen. Anderes Beispiel: Wir addieren zwei negative Zahlen, -4 und -6:

```
    -4      11111010
+   -2    + 11111110
-------------------
    -6    (1)11111000
```

Auch hier kann man (sogar: muß man) das Carry-Bit vernachlässigen. Beide Ergebnisse sind richtig.

Nun wissen Sie alles über die Art, wie unser Rechner mit ganzen Zahlen arbeitet. Probieren Sie mal ein paar Aufgaben aus zur Übung.

Wir verlassen jetzt den Zahlendschungel und widmen uns der Praxis.

17. Der Computer rechnet: ADC, CLC

ADC ist der erste Arithmetik-Befehl des 6502 (und natürlich auch des 6510), den wir kennenlernen. Er bedeutet »ADd with Carry«, also »addiere mit Carry-Bit«. An einem 8-Bit-Beispiel wollen wir uns das mal ansehen. ZAHL1 und ZAHL2 sollen addiert werden. Beide sollen positive 8-Bit-Zahlen sein, die so klein sind, daß kein Überlauf zu erwarten ist. Die ZAHL1 wird in den Akku gegeben:

```
LDA #ZAHL1
```

Wenn wir nun den Befehl

```
ADC #ZAHL2
```

folgen lassen, sorgt die ALU (arithmetisch-logische Einheit, siehe Folge 1) dafür, daß beide Zahlen addiert werden und das Ergebnis im Akku erscheint. ZAHL1 ist dann vom Ergebnis überschrieben worden. An sich ist damit alles erledigt. Weil wir aber häufig wissen wollen, was denn nun bei der Addition herausgekommen ist, speichern wir den Akku-Inhalt noch irgendwo ab mittels »STA Speicherstelle«. Außerdem war da ja noch die Sache mit dem Carry-Bit. Wir haben oben festgestellt, daß bei einer 8-Bit-Addition kein Carry-Bit berücksich-

tigt werden soll. Nun gibt es aber eine ganze Menge von Vorgängen in einem Assembler-Programm, die das Carry-Bit beeinflussen. Man kann eigentlich vor einer Addition nie ganz sicher sein, ob es denn nun 1 oder 0 ist. Weil jedoch ADC auch das Carry-Bit mitaddiert, sollte man dafür sorgen, daß es vor dem Zusammenzählen wirklich gelöscht ist. Dazu gibt es den Befehl CLC was die Abkürzung für »CLear Carry«, also »lösche Carry-Bit« ist. Sei ZAHL1=12 und ZAHL2=7, dann würde unser vollständiges 8-Bit-Additions-Prográmmchen also lauten:

```
1200  CLC
1201  LDA   #$0C
1203  ADC   #$07
1205  STA   1500
```

Sehen wir mal davon ab, daß dieses Programm natürlich unsinnig ist (das kann man ja im Kopf schneller rechnen!), dann erkennen wir: CLC ist ein 1-Byte-Befehl mit impliziter Adressierung, welcher sich nur auf die C-Flagge (also das Carry-Bit) auswirkt. ADC ist in der hier verwendeten Form ein 2-Byte-Befehl und liegt in der »unmittelbar« genannten Adressierung vor. Wie wir oben gesehen haben, kann ADC – je nach Art der Rechnung – auf einige Flaggen wirken: Da wären zunächst natürlich die V-Flagge und die C-Flagge. Dann aber kann beim Auftreten eines gesetzten Bit 7 auch die N-Flagge und beim Überschreiten von $FF eventuell auch die Z-Flagge verändert werden.

Viel interessanter wird unser Mini-Programm schon, wenn man anstelle von

```
1201  LDA   #$0C
```

jetzt die absolute Adressierung verwendet, zum Beispiel

```
1201  LDA   1400
```

Weil das ein 3-Byte-Befehl ist, verschiebt sich natürlich der Rest des Programmes um 1 Byte. So kann man immerhin schon zu unterschiedlichen Inhalten von 1400 den gleichen Betrag addieren.

Am interessantesten allerdings ist die Tatsache, daß auch ADC absolut adressierbar ist. Wir können so zum Beispiel den Inhalt der Speicherzelle 1300 zum Inhalt der Zeile 1400 hinzuzählen und dann das Ergebnis in 1500 ablegen:

```
1200  CLC
1201  LDA   1400
1204  ADC   1300
1207  STA   1500
```

Hier ist der ADC-Befehl dann 3 Byte lang geworden.

Vergessen Sie bitte nicht – das gilt vor allem für die nachfolgenden Rechenoperationen – dann, wenn die Wahrscheinlichkeit besteht, daß der Dezimal-Modus eingeschaltet ist (also die D-Flagge auf 1 gesetzt ist), noch den Befehl CLD vor solche Programme zu stellen.

Solche 8-Bit-Rechnungen kommen recht häufig vor: Wenn man in Schleifen nicht mit mehrfach wiederholten INX (beziehungsweise INY oder INC, DEX, DEY oder DEC) arbeiten will, addiert man eben immer die Sprungweite mittels ADC hinzu. Der Akku kann nicht als Zähler dienen, denn es gibt für ihn keinen Befehl, der dem INX und so weiter vergleichbar wäre, weswegen man ihn – sollte es nötig sein – mittels ADC hochzählt.

Häufiger und in der Praxis bedeutender sind 16-Bit-Rechnungen. Wie Sie sicher noch aus den vorangegangenen Folgen wissen, teilt man so eine 16-Bit-Zahl auf in zwei Byte (das LSB und das MSB). Nehmem wir für unser nachfolgendes Beispiel wieder an, daß die Zahlen so gebaut sind, daß kein Überlauf zu befürchten ist. ZAHL1 hätten wir vorher in die Speicherstellen 1300 (LSB) und 1301 (MSB) gepackt, ZAHL2 liegt in den Zellen 1400 (LSB) und 1401 (MSB). Zunächst wieder die Vorbereitungsmaßnahmen:

```
1200  CLD
1201  CLC
```

Dabei ist CLD nicht immer nötig, wie schon gesagt. Nun addieren wir zuerst die LSBs:

```
1202  LDA   1300
1205  ADC   1400
1208  STA   1500
```

Ein Überlauf kann hier nicht stattgefunden haben, denn das Vorzeichenbit ist ja im MSB als Bit 15 enthalten, wohl aber kann ein Übertrag stattgefunden haben: Das Ergebnis könnte größer als 255 ($FF) gewesen sein. War das der Fall, dann ist jetzt eine 1 im Carry-Bit. Wir addieren nun die MSBs:

```
120   BLDA  1301
120   EADC  1401
1211  STA   1501
```

Egal, was im Carry-Bit stand: Es wurde jetzt hinzuaddiert. Das Ergebnis unserer Rechnung steht nun in 1500 (LSB) und 1501 (MSB). Sehen wir uns das ganze nochmal am Zahlenbeispiel an. Wir addieren die Zahlen 2176 (binär: 0000 1000 1000 0000) und 1009 (binär: 0000 0011 1111 0001). Die Speicherinhalte sind dann:

```
1300  10000000  LSB Zahl1
1301  00001000  MSB
1400  11110001  LSB Zahl2
1401  00000011  MSB
```

Jetzt addieren wir die LSBs:

```
1300          10000000
1400          11110001
Carry                0
-------------------------
1500          01110001
Carry:               1
```

Nun folgt der zweite Teil der Addition mit den MSBs:

```
1301          00001000
1401          00000011
Carry:               1
-------------------------
1501          00001100
```

Damit steht nun das Ergebnis 3185 (binär 0000 1100 0111 0001) säuberlich aufgeteilt in LSB (Speicher 1500) und MSB (Speicher 1501) fest. Das Carry-Bit steht auch nach vollendeter Rechnung noch auf 1, so daß es vor erneuter Addition wieder mit CLC zu löschen ist.

Damit wäre alles über die Addition berichtet. Wie immer in Programmiererkreisen die Empfehlung: üben, üben,....

Wir wenden uns jetzt der gegenläufigen Operation zu: der Subtraktion.

18. Noch mehr Rechnen: SBC, SEC

Daß das Abziehen von Zahlen im Computer durch das Hinzuzählen des Zweierkomplementes geschieht, haben wir mit viel Gehirnschmalzverbrauch schon in vorangegangenen Abschnitten erfahren. Nun sollen Sie die dazu nötigen Befehlsworte des Assemblers kennenlernen. Zunächst einmal ist da SBC. Das heißt »SuBtract with Carry« oder auf deutsch etwa »ziehe unter Berücksichtigung des Carry-Bits ab«. Ebenso wie bei der Addition mit ADC, wirkt das Argument des SBC-Befehls auf den Akku-Inhalt ein – wobei das Ergebnis im Akku landet, diesen also überschreibt. Komplizierter ist hier die Verwendung des Carry-Bits, worauf wir aber nicht detailliert eingehen wollen. (Wen es interessiert: Nachlesen in L.A. Leventhal, »6502 Programmieren in Assembler«, 3. Auflage, München 1983, Seite 3-100). Für uns soll einfach die nicht ganz korrekte Analogie zum »Borgen« bei der Subtraktion ausreichen. Für den Fall, daß ein solches Borgen eintreten muß, sollte auch das dazu nötige Carry-Bit vorhanden

sein (also auf 1 gesetzt sein). Wie Sie sicherlich schon erraten haben, heißt SEC »SEt Carry«, also »setze das Carry-Bit« (auf 1).

Merke: Vor einer Addition immer Löschen des Carry-Bits mit CLC,
vor einer Subtraktion immer Setzen des Carry-Bits mit SEC!

Zwei Beispiele für die Subtraktion sollen das bisher Gesagte erläutern: Zunächst eine 8-Bit-Subtraktion von ZAHL1 (in Speicherzellle 1300) minus ZAHL2 (in Zelle 1400). Das Ergebnis wird nach 1500 geschrieben:

```
1200  CLD
1201  SEC
1202  LDA   1300
1205  SBC   1400
1208  STA   1500
```

SBC kann – wie hier – absolut adressiert werden, aber auch unmittelbar (also zum Beispiel SBC #$40). Der Befehl ist dann im ersten Fall ein 3-, im anderen Fall ein 2-Byte-Befehl. SEC ist ebenso wie vorher schon CLC ein implizit adressierbarer 1-Byte-Befehl.

Das zweite Beispiel ist eine 16-Bit Subtraktion. In den Speichern steht vor dem Aufruf dieser kleinen Routine:

```
1300  ZAHL1  LSB
1301  ZAHL1  MSB
1400  ZAHL2  LSB
1401  ZAHL2  MSB
```

Das Ergebnis soll nach 1500 (LSB) und 1501 (MSB) gebracht werden:

```
1200  CLD
1201  SEC
1202  LDA   1300
1205  SBC   1400
1208  STA   1500
```

Jetzt sind die beiden LSBs voneinander abgezogen und die Differenz abgespeichert als LSB des Ergebnisses.

```
120B  LDA   1301
120E  SBC   1401
1211  STA   1501
```

Damit ist die Aufgabe beendet. Auch die MSBs sind subtrahiert und das MSB des Ergebnisses steht in 1501.

SBC beeinflußt die gleichen Flaggen wie der Befehl ADC.

19. Ein Programmprojekt

Damit die so kennengelernten Arithmetik-Befehle nicht so trocken auf weiter Flur stehen, wollen wir nun ein Programm entwickeln, aus dem zweierlei zu lernen ist:
1) Die Anwendung bisher gelernter Befehle und
2) ein häufig angewendetes Verfahren, Assemblerprogramme in Basic-Programme einzubinden.

Besonders dieser zweite Aspekt scheint noch vielen Lesern unklar zu sein (das zeigen mir Zuschriften). Es gibt eine ganze Reihe von Möglichkeiten, zum Einbau von Assembler-Routinen in Basic-Programme; die werden wir alle nach und nach kennenlernen. Von Ihnen wurde der SYS-Befehl sicherlich schon häufig angewendet (zum Beispiel für SYS 58640 und vorherigem POKE214,Zeile und POKE211, Spalte zum Setzen des Cursors an die Stelle Zeile, Spalte). Damit haben Sie ein Maschinenprogramm aufgerufen, das im System unseres Computers schon enthalten ist. 58640 ist die Startadresse des Programmes und man kann diesen SYS-Befehl eigentlich wie eine Art »GOTO Maschinenprogramm-Startadresse« ansehen. Nichts hindert uns also, auf diese Weise eigene Assembler-Programme anzuspringen! Das Problem liegt nun nur noch darin, wie man Parameter, die unsere Maschinenroutine benötigt, übergeben kann. Eine offensichtliche – aber leider auch relativ langsame – Methode ist das POKEn der Werte im LSB/MSB-Format in die Speicherzellen, aus denen sie sich unser ML-Programm dann abholt. Wir wollen dieses Verfahren nun an einem Programmbeispiel verwenden.

Eine arithmetische Reihe werden viele von Ihnen schon kennen. Wenn man A als erstes Glied, D als Differenz und N als die Anzahl der Glieder bezeichnet, dann ist die Summe einer solchen Reihe:

$$S=A+(A+D)+(A+2*D)+.....+(A+(N-1)*D)$$

Ein Beispiel ist die Summe der ersten zehn ganzen Zahlen:

$$S=1+2+3+4+5+6+7+8+9+10$$

Hier ist A=1, D=1 und N=10. Daß die Summe S im Beispiel 55 ist, kann man schnell berechnen, was aber, wenn wir wesentlich mehr als nur zehn Glieder haben? Es gibt natürlich auch Formeln zur Berechnung von S. Aber eigentlich ist es ganz reizvoll, ohne solche Formeln den Computer die Summe bilden zu lassen. Wir bauen also ein Programm zur Berechnung der Summe der ersten N ganze Zahlen, wobei N frei gewählt werden kann. Das Ergebnis soll eine 16-Bit-Zahl sein, also nicht größer als 32767. Das beschränkt uns bei N auf Werte von 1 bis 255 (Warum, können Sie ja mal mit dem fertigen Programm ausprobieren). N benötigt also nur 1 Byte Speicherplatz und soll in $1300 abrufbar sein. A soll 1 sein ebenso wie D. Für eventuelle Programmänderungen ist es aber sinnvoll, A und D als 16-Bit-Zahlen aufzubewahren und zwar in $1310/1311 (A in LSB/MSB-Format) und in $1320/1321 (D im gleichen Format). Das Ergebnis soll in $1400/1401 zu finden sein. Das Maschinenprogramm legen wir nach $1200.

Zuerst kümmern wir uns um das Basic-Aufrufprogramm:

Zu diesem Programm gibt es nur noch zu bemerken, daß die Zahlen bei POKE, PEEK oder SYS die Dezimalwerte unserer oben gewählten Adressen sind.

Nun endlich zum Assemblerprogramm. Sehen Sie sich dazu bitte das Flußdiagramm im Bild 12 an.

Wir bereiten den Ablauf vor, indem wir aus $1300 die Anzahl der Glieder ins X-Register laden und zur Vorbereitung der Addition das Carry-Bit löschen. Schalten Sie also bitte den SMON ein und tippen Sie A1200 <RETURN>. Es erscheint die Startadresse 1200. Jetzt können Sie Zeile für Zeile das Assembler-Programm eingeben (nach jeder Zeile ein RETURN, das die nächste Zeilennummer erzeugt):

```
1200  LDX   1300
1203  CLC
```

Die nächsten sechs Zeilen summieren jeweils das neueste Glied zur bis dahin erzeugten Summe. Jetzt zu Beginn ist

```
10  REM **AUFRUF SUMME ARITHMETISCHE REIHE**
20  POKE5120,0:POKE5121,0:REM ERGEBNISSPEICHER AUF
    NULL
30  PRINTCHR$(147)CHR$(17)CHR$(17)
40  INPUT"ANZAHL DER GLIEDER N=";N
50  IFN<1 OR N>255 THEN PRINT CHR$(17)"1<=N
    <=255":GOTO40
60  POKE4864,N:REM EINSPEICHERN VON N
70  POKE4880,1:POKE4881,0:POKE4896,1:
    POKE4897,0:REM EINSPEICHERN VON A UND D
80  SYS4608:REM AUFRUF UNSERES MASCHINEN-
    PROGRAMMES
90  M=PEEK(5121):L=PEEK(5120):REM AUSLESEN DES
    ERGEBNISSES
100 E=256*M+L:PRINTCHR$(17)CHR$(17)
110 PRINT"DIE SUMME DER ERSTEN "N" GANZEN ZAHLEN
    IST:":PRINTE
120 END
```

$1400/1401 noch leer und in $1310/1311 steht noch das Anfangsglied A=1. Später mit Durchlaufen der Schleife, steht in $1400/1401 immer die bis dahin gebildete Summe und in $1310/1311 das letzte Glied der Reihe. Es handelt sich um die oben kennengelernte 16-Bit-Addition:

```
1204   LDA    1400
1207   ADC    1310
120    ASTA   1400
```

Das neue LSB ist berechnet und in $1400 geschrieben.

```
1200   LDA    1401
1210   ADC    1311
1213   STA    1401
```

Das war nun noch das neue MSB. Als nächstes berechnen wir das momentan letzte Glied der Reihe durch Addieren von D zum alten letzten Glied. Das entspricht dem Basic-Befehl A=A+D in einer Schleife. Dies ist eine neue 16-Bit-Addition, weshalb wir wieder CLC vorgeben müssen:

```
1216   CLC
1217   LDA    1310
121    AADC   1320
121    DSTA   1310
```

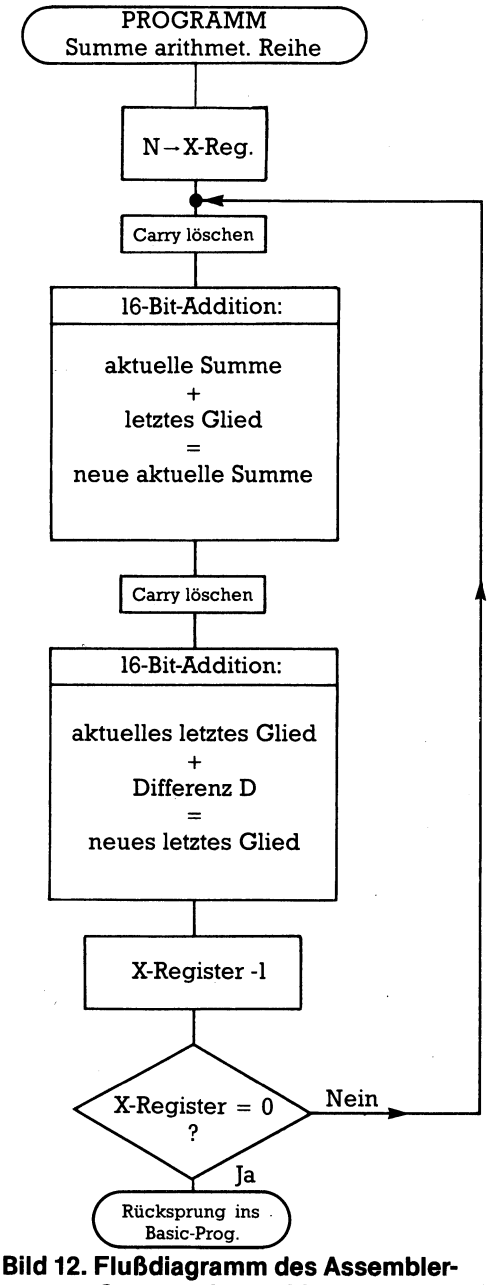


Bild 12. Flußdiagramm des Assemblerprogrammes »Summe einer arithmetischen Reihe«

Das war wieder das LSB. Nun zum MSB:

```
1220   LDA    1311
1223   ADC    1321
1226   STA    1311
```

Wir zählen nun das X-Register um 1 herunter und prüfen, ob es schon Null geworden ist, ob also schon alle N-Glieder summiert worden sind:

```
1229   DEX
122    ABNE   1203
```

Wenn noch nicht alle Glieder berechnet und summiert sind, kehren wir an den Schleifenanfang zurück. Ansonsten springen wir zurück ins Basic-Aufrufprogramm:

```
122    CRTS
```

Wenn Sie beide Programme eingetippt haben, dann speichern Sie sie vorsichtshalber ab (das Assemblerprogramm mit dem S-Befehl des SMON). Beim neuen Einladen brauchen Sie den SMON nicht mehr. Nach dem Laden unseres Maschinenprogrammes (mit ,8,1 bei Diskette oder ,1,1 bei Kassette) geben Sie NEW <RETURN> ein, damit die Zeiger vor dem Einladen des Basic-Programmes wieder auf Normalwerte gesetzt werden. Zwischen dem dann eingeladenen Basic-Programm und unserer Assembler-Routine ist genug Platz. Sollten Sie aber irgendwann mal das Basic-Programm vergrößern, schützen Sie bitte unseren Bereich ab $1200.

Unser Assembler-Beispiel ist so aufgebaut, daß auch A und D variabel gehalten sind. Sie müßten dann nur noch Eingabemöglichkeiten im Basic-Programm schaffen und anstelle der Werte 1 oder 0 in Zeile 70 die LSBs und MSBs der von Ihnen eingegebenen Größen A und D einPOKEn. Auf diese Weise sind dann beliebige ganzzahlige, arithmetische Reihen berechenbar, wie zum Beispiel S=7+10+13+16+... und so weiter. Das überlasse ich Ihrer Basic-Programmierfertigkeit. Nur eines noch: Sie müssen darauf achten, daß die Summe S nicht größer als 32767 wird. Ihrer Phantasie sind – wie immer in diesem Metier – keine Grenzen gesetzt. Sie könnten sich ja mal überlegen, wie man größere Summen zulassen kann (wer sagt denn, daß wir Zahlen immer nur in 2 Byte darstellen dürfen?). Oder Sie könnten sich überlegen, welches eindeutige Merkmal auftritt, sobald der Maximalwert überschritten wird (ein Tip: Lesen Sie doch mal den Abschnitt über die V-Flagge nach).

20. Die Branch-Befehle

Der 6502 (und auch der damit identische 6510) kennt acht bedingte Verzweigungen, von denen wir bisher BNE schon verwendet haben. Alle diese Branch-Befehle (von branch = verzweigen) prüfen Flaggen des Statusregisters.

BNE und BEQ beziehen sich auf die Z-Flagge, die anzeigt, ob im Verlauf der letzten Operation eine Null aufgetreten ist. Ist das der Fall, steht in der Z-Flagge eine 1. BNE verzweigt zur angegebenen Adresse, wenn in der Z-Flagge eine 0 enthalten ist. BEQ (»Branch if EQual zero« = »verzweige, wenn gleich Null«) tut das dann, wenn die Z-Flagge auf 1 gesetzt ist. Da muß man etwas aufpassen, daß man sich nicht vertut!

BCC und BCS haben ihre Aufmerksamkeit auf die C-Flagge, also das Carry-Bit gerichtet. BCC kommt vom englischen »Branch if Carry Clear«, was heißt: »verzweige, wenn das Carry-Bit gelöscht ist«. Ein gesetztes Carry-Bit (also Inhalt=1) veranlaßt BCS (»Branch if Carry Set« = verzweige, wenn das Carry-Bit gesetzt ist) zum Sprung an die angegebene Adresse.

Diese vier bedingten Verzweigungen sind an sich die bedeutsamsten und am häufigsten verwendeten Branch-Befehle. Man kann wohl getrost sagen, daß über 90% der von Programmierern verwendeten bedingten Sprünge, damit absolviert werden. R. Mansfield warnt sogar ausdrücklich in

seinem Buch »Machine language for beginners«, einem in den USA sehr verbreitetem Werk, vor der Verwendung der Befehle BPL und BMI!

Dafür liegt absolut kein einsehbarer Grund vor. Viele programmtechnischen Aufgabenstellungen lassen sich elegant und leicht mit BPL, BMI, BVS und BVC lösen. Man muß nur wissen, wie sie funktionieren und - da liegt vermutlich der Hund begraben - man muß auch die Art kennen, wie Zahlen vom Computer behandelt werden. Genau das aber wissen wir und deswegen sollten wir diese Kenntnis für uns auch nutzen. Also ohne Scheu heran an die verfehmten Befehle!

BMI und BPL (»Branch on MInus« = »verzweige, wenn negativ« und Branch on PLus« = »verzweige, wenn positiv«) hängen mit der Negativ-Flagge N zusammen. Das Rätsel dieser Flagge konnte in den vorangegangenen Folgen gelöst werden: Immer dann, wenn bei einer Operation eine Zahl auftrat, deren Bit 7 eine 1 war, wurde die N-Flagge auf 1 gesetzt. Wir wissen jetzt, daß dieses Bit bei 8-Bit-Zahlen das Vorzeichenbit ist. Bit 7 sagte uns bei einer 1, daß eine negative Zahl im Zweierkomplement-Format vorliegt oder aber überhaupt ein Speicherzelleninhalt vorhanden ist, der größer als 0111 1111 = 127 ist. BMI führt zum Sprung in diesem Fall, weil die N-Flagge auf 1 steht. Andernfalls führt BPL zur Verzweigung.

Ebenso einfach sind BVS und BVC zu erklären: Sie beziehen sich auf die V-Flagge, unsere rote Ampel, die Überlauf bei Rechenoperationen anzeigt. Kann es was bequemeres geben zur Behandlung solcher Fehlrechnungen als ein »Branch on oVerflow Set« = »verzweige, falls die Überlauf-Flagge gesetzt (=1) ist« mit BVS? Oder anders herum bei BVC »Branch on oVerflow Clear« = »verzweige bei freier Überlauf-Flagge«. Wenn man - wie Sie jetzt - weiß, unter welchen Umständen diese V-Flagge auf 1 gesetzt wird, sollte man ohne Skrupel BVS und BVC ausgiebig benutzen. Man könnte damit zum Beispiel programmieren, daß die Rechengenauigkeit automatisch von 16-Bit auf 24- oder 32- (oder wie es gerade beliebt) Bit gesteigert wird, ohne daß man sich bei jeder Programmaufgabe Gedanken über das größtmögliche Ergebnis machen muß. Dazu aber ein andermal mehr.

Alle hier vorgestellten Branch-Befehle sind ebenso wie BNE 2-Byte-Befehle, was an der speziellen Art der Adressierung liegt: Der relativen Adressierung. Tabelle 3 zeigt eine Übersicht der neuen Befehle aus den letzten fünf Kapiteln.

Befehlswort	Adressierung	Byteanzahl	Code hex	Code dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
ADC	unmittelbar	2	69	105	2	N,V,Z,C
	absolut	3	6D	109	4	
CLC	implizit	1	18	24	2	0→ C
SBC	unmittelbar	2	E9	233	2	N,V,Z,C
	absolut	3	ED	237	4	
SEC	implizit	1	38	56	2	1 → C
BEQ	relativ	2	F0		2	keine Änderung
BCC	relativ	2	90		2	keine Änderung
BCS	relativ	2	B0		2	keine Änderung
BMI	relativ	2	30		2	keine Änderung
BPL	relativ	2	10		2	keine Änderung
BVC	relativ	2	50		2	keine Änderung
BVS	relativ	2	70		2	keine Änderung
					+1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	

Tabelle 3: Die 11 neuen Befehle

21. Die relative Adressierung

Als wir den BNE-Befehl das erstemal verwendet haben, stellten wir fest, daß zum Beispiel BNE 1200 nicht — wie eigentlich zu erwarten war — ein 3-Byte-Befehl, sondern ein 2-Byte-Befehl ist. Damals mußten wir uns mit der Bemerkung zufrieden geben, es läge an der besonderen Art der Adressierung, nämlich der relativen Adressierung. Relativ bedeutet ja »bezogen auf etwas«. Wenn wir also beispielsweise BNE 1200 schreiben, liegt es nur an der Benutzerfreundlichkeit des SMON und vieler anderer Assembler, daß dieser die so geschriebene absolute Adresse 1200 in die richtige Form, nämlich die relative umrechnet. In Wahrheit verlangt der 6502 (und natürlich ebenso der 6510) eine Angabe darüber, wieviele Bytes nach vorne oder hinten im Programm er zur weiteren Programmverarbeitung springen (verzweigen) soll. Es gilt nun also, zwei Fragen zu klären:

1. Relativ wozu wird gesprungen und
2. Wie berechnet sich die Angabe, um wieviele Bytes nach vorne oder hinten im Programm der Sprung vollzogen werden soll.

Zur Klärung verwenden wir ein hypothetisches Programmsegment mit einem Sprungbefehl und sehen uns das Disassemblerlisting an:

```
2000  AD 0030   LDA   3000
2003  F0 05     BEQ   200A
2005  A9 00     LDA   #00
2007  8D 0030   STA   3000
200A  60        RTS
```

Dieses Programm-Teilchen lädt den Inhalt der Speicherstelle 3000 in den Akku, überprüft dann, ob dieser Inhalt Null ist und verzweigt beim Vorliegen der Null zum Rücksprung (RTS). Ist der Inhalt von 3000 nicht Null, dann wird 3000 auf Null gesetzt. 3000 könnte zum Beispiel eine Flagge sein.

Der Pfad, dem der Computer bei der Abarbeitung des Programmes folgt, wird durch den Programmzähler vorbereitet. Dieser ist dann, wenn der BEQ-Befehl an der Reihe ist, schon einen Schritt weiter, nämlich im Programmzähler steht dann die Adresse 2005.

Relativ zu dieser Adresse hat dann der Sprung zu erfolgen. Zum Inhalt des Programmzählers muß also die Sprungweite (auch häufig Offset genannt) addiert werden. Soweit zur Frage 1.

Zur Klärung von Frage 2 listen wir uns mal Byte für Byte unser Programm auf:

Byte		Inhalt	Bedeutung
2000		AD	LDA
2001		00	LSB von 3000
2002		30	MSB von 3000
2003		F0	BEQ
2004		05	Offset
2005		A9	LDA #
2006	1	00	
2007	2	8D	STA
2008	3	00	LSB von 3000
2009	4	30	MSB von 3000
200A	5	60	RTS

Neben der Byte-Nummer ist noch die Entfernung zu 2005 geschrieben. Daraus ist deutlich zu erkennen, daß die Sprungweite, die zum Programmzähler addiert wird, 05 sein muß, wenn der Sprung zum RTS erfolgen soll. Für Vorwärts-Verzweigungen gilt also: Von der Adresse des Befehls an, der auf den Branch-Befehl folgt, zählt man die Byte-Anzahl bis zum Sprungziel. Das Ergebnis ist der Offset.

Nun gibt es genauso häufig Rückwärts-Sprünge. In den bisher gezeigten Programmen sind sie mehrmals aufgetreten. Wie berechnet man den Offset in diesen Fällen? Sehen wir uns wieder das Disassembler-Listing eines solchen Programmsegmentes an:

```
1000   A2 00   LDX #00
1002   E8      INX
1003   D0FD    BNE 1002
1005   00      BRK
...
```

Dieses Prögrämmchen tut nichts anderes, als das vorher auf Null gesetzte X-Register hochzuzählen, bis es über 255 läuft (dann tritt ja wieder 0 auf!). Solange der Inhalt des X-Registers ungleich Null ist, erfolgt ein Sprung zurück bis zur INX-Anweisung in Zeile 1002. Erst wenn die Null durch den Überlauf aufgetreten ist, endet das Programm mit einem BRK in Zeile 1005.

Wir wissen schon, daß der Programmzähler beim Verarbeiten des BNE-Befehls auf 1005 steht. Sehen wir uns auch dieses Programm Byte für Byte an:

Byte		Inhalt	Bedeutung
1000		A2	LDX #
1001		00	
1002	3	E8	INX
1003	2	D0	BNE
1004	1	FD	Offset
1005		00	BRK

Wieder ist neben der Bytenummer die Entfernung vom aktuellen Programmzählerstand angegeben. Wir müssen also vom Inhalt des Programmzählers 3 abziehen, um zum INX-Befehl in Byte 1002 zu gelangen. Das kennen wir aber schon aus den vergangenen Ausgaben: Wenn der Computer eine Zahl abzieht, dann addiert er das Zweierkomplement dieser Zahl. Hier soll nun 3 subtrahiert werden. Wir berechnen das Zweierkomplement:

3 = 0000 0011 (binär)

Das Einerkomplement davon ist:

1111 1100

Dann wird eine 1 addiert

1111 1101

Dies ist das Zweierkomplement. In hexadezimal ausgedrückt heißt diese Zahl $FD und ist unser Offset. Für Rückwärts-Verzweigungen gilt also: Von der auf die Branch-Anweisung folgenden Speicherstelle an zählt man die Bytes zurück bis zum Sprungziel. Das Zweierkomplement der sich dadurch ergebenden Byte-Anzahl ist der Offset.

Das sieht reichlich kompliziert aus, aber zum einen haben Sie ja einen ganz freundlichen Assembler und nur in seltenen Notfällen müssen Sie den Offset berechnen. Zum anderen gibt es noch eine Faustregel, mit der man sich das ganze vereinfachen kann. Die soll durch folgendes Schema erläutert werden:

Byte	Inhalt	Offset
...		
1995		F9
1996		FA
1997		FB
1998		FC
1999		FD
2000	BNE	FE
2001	Offset	FF
2002	Programm-zählerstand	
2003		01
2004		02
2005		03
..		

Bei Vorwärtssprüngen ist ohnehin alles klar: Bei einem Sprung nach Adresse 2005 müßte man in vorliegendem Fall einen Offset von 03 eingeben. Bei Rückwärts-Verzweigungen zählt man einfach von $FF an rückwärts bis zur Zieladresse. Eine Verzweigung nach 1996 würde im vorliegenden Fall also einen Offset von $FA erfordern.

Eine Einschränkung der relativen Adressierung können Sie nun auch sofort verstehen, wenn Sie an Zweierkomplementzahlen denken: Der Offset belegt ein Byte. Die größte positive Zahl in einem Byte ist

0111 1111 = +127 = $7F

und die kleinste negative Zahl ist

1000 0000 = —128 =($80)

Es sind keine größeren Vorwärts-Verzweigungen als um 127 Byte möglich, weil in diesem Fall ein Offset größer als $7F, also mit einem Bit 7 gleich 1 nötig wäre, was aber wieder als negative Zweierkomplementzahl verstanden und einen Rückwärtssprung verursachen würde. Ähnliches gilt anders herum: Es ist kein weiterer Rücksprung als um 128 Byte möglich, weil das im Offset zum gelöschten Bit 7 führen würde, also zu einem Offset kleiner als $80, was wiederum anstelle des Rücksprunges eine Vorwärts-Verzweigung herbeiführen würde.

Darauf sollte man achten beim Erstellen eines Assembler-Programmes, daß man nie weitere Rückwärtssprünge als um 128, beziehungsweise Vorwärtssprünge um 127 Byte verlangt. Auch wenn man im Assembler gar nicht auf relative Adressierung Rücksicht nehmen muß, weil der Assembler sich mit den Absolutadressen begnügt, sollte man wissen, daß zum Beispiel folgende Zeile aufgrund dieser Einschränkung nicht möglich ist:

```
3000 BNE 1000
```

Die meisten Assembler reagieren auf solch eine Zeile mit einer Fehlermeldung (beim Hypra-Ass mit »Branch too far«) oder so wie der SMON, der klammheimlich die Programmstartadresse statt 1000 einsetzt. Aber es ist doch ärgerlich, wenn man auf dem Papier ein Programm fertig hat und erst beim Eintippen feststellt, daß der Computer das so nicht haben will.

22. Zeropage-Adressierung

Weil wir nun gerade mit der Adressierung so schön in Schwung sind, stelle ich Ihnen noch eine andere vor: Die Adressierung der Zeropage. Was ist die Zeropage? Auf deutsch heißt das Nullseite. Am besten versteht man das, wenn man sich in Erinnerung ruft, wie Adressen in unserem Computer verwaltet werden. Da haben wir doch ein LSB (Least Significant Byte) und ein MSB (Most Significant Byte), zum Beispiel $1F 04 (mit 1F als MSB und 04 als LSB). Nun hat unser C 64 65535 Adressen von $0000 bis $FFFF. Bei den ersten 256 Adressen von $0000 bis $00FF ist das MSB $00. Man nennt so einen 256-Byte-Block eine Seite (engl. page). Weil hier für alle Adressen dieser ersten Seite des MSB Null ist heißt sie Nullseite = Zeropage. Messerscharf werden Sie schließen, daß man die Seite mit den MSBs $01 als erste Seite bezeichnet, die mit den MSBs $02 als zweite Seite und so weiter.

Wenn wir nun zum Beispiel den Akku mit dem Inhalt der Zeropageadresse $00FA laden wollen, dann könnten wir schreiben:

```
3000 LDA 00FA
```

Unser Mikroprozessor versteht uns aber auch, wenn wir nur schreiben:

```
3000 LDA FA
```

Das ist sie, die Zeropage-Adressierung. Anstelle eines 3-Byte-Befehls ist das jetzt ein 2-Byte-Befehl, was Speicherplatz und vor allem Rechenzeit einspart. Auf diese Weise kann man von den bisher kennengelernten Befehlen folgende adressieren:
LDA, LDX, LDY, STA, STX, STY, INC, DEC, ADC und SBC

Sie können sich merken, daß man (bis auf zwei Ausnahmen, die wir noch kennenlernen werden) alle absolut adressierbaren Befehle auch Zeropage-absolut anwenden kann. Genauere Angaben über die Codes, die Ausführungszeiten und die Beeinflussung der Flaggen (letztere ist identisch mit der absoluten Adressierung) entnehmen Sie bitte der angefügten Tabelle 4.

Zum Thema Geschwindigkeit: Wenn Sie die benötigten Taktzyklen von absolut und von 0-absolut adressierten Befehlen in den Tabellen miteinander vergleichen, werden Sie jeweils einen Unterschied von einem Zyklus feststellen. Das mag Ihnen läppisch vorkommen. Bedenken Sie aber, daß Sie sehr häufig Schleifen programmieren müssen, die mehrere 100 Mal durchlaufen werden, die vielleicht als oft zu verwendende Unterprogramme dienen... Sie werden bald feststellen, daß da schnell beachtliche Zeitunterschiede auftreten können: Für zeitkritische Programme ist die Verwendung der Zeropage-Adressierung dringend geboten.

Dieser Tatsache waren sich leider auch die Schöpfer unseres Betriebssystems und des Basic-Interpreters voll bewußt. Die Zeropage ist nahezu randvoll mit Speicherstellen, in denen sich beide Programmkomplexe tummeln. Fast jede Kernel- und Interpreter-Routine notiert sich irgendwelche Werte auf der Seite Null. Das macht es uns als Assembler-Programmierer nicht gerade leicht, die Zeropageadressierung zu verwenden, wenn wir außerdem den Interpreter oder das Betriebssystem benutzen wollen. Es kann geradezu katastrophale Folgen haben, einige Zeropage-Adressen zu überschreiben. Andere werden ständig neu beschrieben durch das Betriebssystem oder den Interpreter, was unseren eigenen — vielleicht gerade in so einer Speicherzelle gelagerten — Zwischenwerten den Garaus machen würde. Man sollte sich also die ersten 256 Speicherstellen ganz genau ansehen, bevor man sie adressiert oder aber auf das Betriebssystem und den Basic-Interpreter verzichten. Ersteres erleichtern uns Tabellen der Speicherbelegung (zum Beispiel Babel, Krause, Dripke »Das Interface Age Systemhandbuch zum Commodore 64«, Interface Age Verlag, oder »Das Commodore 64 Buch, Band 4, Ein Leitfaden für Systemprogrammierer«, Markt und Technik Verlag) und auch die Serie von Dr. Helmut Hauck »Memory Map mit Wandervorschlägen«, die seit Ausgabe 11/84 im 64'er erscheint.

Befehlswort	Adressierung	Byteanzahl	Code Hex	Code Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	0-Page, abs.	2	A5	165	3	N,Z
LDX	0-Page, abs.	2	A6	166	3	N,Z
LDY	0-Page, abs.	2	A4	164	3	N,Z
STA	0-Page, abs.	2	85	133	3	—
STX	0-Page, abs.	2	86	134	3	—
STY	0-Page, abs.	2	84	132	3	—
INC	0-Page, abs.	2	E6	230	5	N,Z
DEC	0-Page, abs.	2	C6	198	5	N,Z
ADC	0-Page, abs.	2	65	101	3	N,V,Z,C
SBC	0-Page, abs.	2	E5	229	3	N,V,Z,C
CMP	unmittelbar	2	C9	201	2	N,Z,C
	absolut	3	CD	205	4	
	0-Page, abs.	2	C5	197	3	
CPX	unmittelbar	2	E0	224	2	
	absolut	3	EC	236	4	
	0-Page, abs.	2	E4	228	3	
CPY	unmittelbar	2	C0	192	2	
	absolut	3	CC	204	4	
	0-Page, abs.	2	C4	196	3	

Tabelle 4: Kenndaten der neuen Befehle und Adressierungen

Ohne Hemmungen dürfen wir nur die Speicherstellen (jedenfalls beim C 64) $02 und $FB bis $FE nutzen. Weil das doch recht mickrig ist, hat jeder Assembler-Programmierer spezielle Tips, welche Zellen er noch mit welchen Vorsichtsmaßnahmen benutzt. Wenn man bestimmte Routinen aus dem Betriebssystem oder dem Interpreter nicht aufruft, bleiben dazugehörige Zeropageadressen unbeeinflußt und sind dann für eigene Zwecke nutzbar. Manchmal ist es notwendig, den alten Zustand einer Adresse nach Beendigung eigener Programme wieder herzustellen, manchmal nicht. Interessant und viel beschrieben in allen möglichen Zeitschriften, Büchern etc. ist die Möglichkeit, die Notizen, die sich das Betriebssystem oder der Interpreter auf der Zeropage macht, zu verändern. Im Prinzip schreibt man damit kleine Teile dieser Großprogramme um oder variiert Tabellenteile davon. Wie schon Dr. Hauck in seiner Serie sagt, geschieht das im Rahmen der »Tricks« mit irgendwelchen POKEs mehr oder weniger blind, weshalb auch bevorzugt Abstürze des Computers dabei festzustellen sind. Warum Abstürze? Na, stellen Sie sich mal ein von Ihnen geschriebenes Programm vor — zum Beispiel das aus Kapitel 19 zur Berechnung der Summe einer arithmetischen Reihe — und POKEn Sie dann anstelle irgendeines Befehlscodes, der dorthin gehört, jetzt eine 0 (also ein BRK) hinein. Die Wirkung dürfte ähnlich sein. Wenn man allerdings die Funktion der betreffenden Speicherstelle genau kennt, lassen sich recht nützliche Änderungen hervorrufen, wie zum Beispiel die Schutz-POKEs für den Basic-Speicher durch Verändern der Adressen $33, $34, $37 und $38.

Wir werden im folgenden immer dann, wenn wir mit Zeropage-Adressierung arbeiten oder Routinen des Betriebssystems oder Interpreters untersuchen, spezielle Stellen der Nullseite kennenlernen.

Vorhin hatte ich noch angedeutet, daß man dann die Zeropage fast vollständig nutzen könne, wenn man auf den Basic-Interpreter und das Betriebssystem verzichtet. Das ist tatsächlich möglich. Nur wird man dann erstaunt feststellen, wieviel Arbeit uns die computerinterne Software abnimmt oder anders herum: Viele bislang selbstverständliche Dinge werden wir dann plötzlich selbst programmieren müssen, und das kann ein hartes Brot sein!

Als Beispiel für ein Programm, das nicht nur die Zeropageadressierung verwendet, sondern sogar selbst komplett in der Zeropage steht, werden wir uns die CHRGET-Routine ansehen. Eine Klasse von Befehlen, die dort angewendet wird, die Vergleichsbefehle, soll zuvor noch gezeigt werden.

23. Die Vergleichsbefehle: CMP, CPX, CPY

Vergleichen heißt in englischer Sprache »to compare«, woraus Sie unschwer erkennen können, woher die Bezeichnung CMP und die CPs in CPX beziehungsweise CPY kommen. Verglichen wird jeweils der Akku-Inhalt (bei CMP), der Inhalt des X- (bei CPX) oder des Y-Registers (bei CPY) mit Daten, die der Compare-Befehl adressiert. Einige Beispiele werden Ihnen das klarer machen:

```
CMP #FF
```

vergleicht den Akku-Inhalt mit der Zahl $FF. Hier liegt die unmittelbare Adressierung vor, die ebenso für CPX und CPY verwendbar ist. Außerdem ist das dann ein 2-Byte-Befehl.

```
CPX 3000
```

vergleicht den Inhalt des X-Registers mit dem Inhalt der Spei-

cherstelle $3000. Die absolute Adressierung ist also auch anwendbar (natürlich auch für CMP und CPY). Der Compare-Befehl besteht so aus 3 Byte.

CPY A8

vergleicht den Inhalt des Y-Registers mit dem Inhalt der Zeropagestelle $A8. Diese soeben frisch gelernte Zeropage-Adressierung ist bei allen drei Vergleichsbefehlen möglich und macht aus ihnen 2-Byte-Befehle.

Für CPX und CPY sind das alle Möglichkeiten der Adressierung. CMP erlaubt weitere, die wir noch kennenlernen werden. Nun interessiert uns natürlich noch, wie das Vergleichsergebnis zu erhalten ist! Bei diesen Befehlen geschieht merkwürdiges: Die Vergleichsdaten werden vom Inhalt des Akkus (beziehungsweise X- oder Y-Registers) abgezogen, aber: Weder wird dieser Inhalt noch werden die adressierten Daten verändert! Der Trick ist, daß drei Flaggen das Ergebnis anzeigen: Die Negativ-Flagge N, die Null-Flage Z und das Carry-Bit C. Diese Anzeige geschieht so: (Bild 13)

FLAGGE	Akku, X, Y > DATEN	Akku, X, Y = DATEN	Akku, X, Y < DATEN
N	0 oder 1	0	1 oder 0
Z	0	1	0
C	1	1	0

Bild 13. Flaggen bei den Vergleichsbefehlen

lsn / msn $		$\overline{\$}$ bin. binär	0	1	2	3	4	5	6	7
			0000	0001	0010	0011	0100	0101	0110	0111
0	0	000	NUL	DLE	SP	0	@	P		p
			NULL	DLE	SP	0	@	P	CHR$(96)	CHR$(112)
1	0	001	SOH	DC1	!	1	A	Q	a	q
			SOH	DC1	!	1	A	Q	CHR$(97)	CHR$(113)
2	0	010	STX	DC2	"	2	B	R	b	r
			STX	DC2	"	2	B	R	CHR$(98)	CHR$(114)
3	0	011	ETX	DC3	#	3	C	S	c	s
			ETX	DC3	#	3	C	S	CHR$(99)	CHR$(115)
4	0	100	EOT	DC4	$	4	D	T	d	t
			EOT	DC4	$	4	D	T	CHR$(100)	CHR$(116)
5	0	101	ENQ	NAK	%	5	E	U	e	u
			ENQ	NAK	%	5	E	U	CHR$(101)	CHR$(117)
6	0	110	ACK	SYN	&	6	F	V	f	v
			ACK	SYN	&	6	F	V	CHR$(102)	CHR$(118)
7	0	111	BEL	ETB	'	7	G	W	g	w
			BEL	ETB	'	7	G	W	CHR$(103)	CHR$(119)
8	1	000	BS	CAN	(	8	H	X	h	x
			BS	CAN	(	8	H	X	CHR$(104)	CHR$(120)
9	1	001	HT	EM	)	9	I	Y	i	y
			HT	EM	)	9	I	Y	CHR$(105)	CHR$(121)
A	1	010	LF	SUB	*	:	J	Z	j	z
			LF	SUB	*	:	J	Z	CHR$(106)	CHR$(122)
B	1	011	VT	ESC	+	;	K	[	k	{
			VT	ESC	+	;	K	[	CHR$(107)	CHR$(123)
C	1	100	FF	FS	,	<	L	\	l	|
			FF	FS	,	<	L	£	CHR$(108)	CHR$(124)
D	1	101	CR	GS	—	=	M	]	m	}
			CR	GS	—	=	M	]	CHR$(109)	CHR$(125)
E	1	110	SO	RS	·	>	N	↑	n	~
			SO	RS	·	>	N	↑	CHR$(110)	CHR$(126)
F	1	111	SI	US	/	?	O	—	o	DEL
			SI	US	/	?	O	←	CHR$(111)	CHR$(127)

Bild 14: ASCII-Code (jeweils oben) und Commodore-ASCII (msn = most significant nibble; lsn = least significant ni.

1) Der Registerinhalt (Akku, X-, Y-Register) ist größer als die Vergleichsdaten:
Dann ist das Carry-Bit = 1, die N- und die Z-Flagge =0.
2) Der Registerinhalt ist gleich den Vergleichsdaten:
Dann sind Carry- und Z-Flagge = 1, die N-Flagge = 0.
3) Der Registerinhalt ist kleiner als die Vergleichsdaten:
Die N-Flagge ist dann =1, Carry- und Zero-Flagge sind 0.

Damit Sie die Übersicht behalten können, ist in Bild 13 das ganze als Schema gezeigt.

Sie werden sich vermutlich schon denken können, wie der Hase weiterläuft: Mit den Verzweigungsbefehlen prüfen wir die Flaggen und springen die gewünschten weiteren Programm-Routinen an.

Die Kombination der Compare-Befehle mit den Verzweigungsoperationen wird Ihnen im weiteren Verlauf dieses Kurses noch ganz geläufig werden. Ein Beispiel sehen Sie nachher ebenfalls in der CHRGET-Routine. Leider muß ich Sie immer noch etwas vertrösten, denn mit Verstand begreifen läßt sich diese Routine nur dann, wenn man etwas mehr über die Codierung von Zeichen weiß. Deswegen werden wir uns nun noch mit dem ASCII-Code und dem Commodore-ASCII herumschlagen.

24. Zeichencodierung mit dem ASCII- und dem Commodore-ASCII-Code

ASCII ist die Abkürzung von »American Standard Code for Information Interchange« und das heißt auf deutsch »amerikanischer Standard-Code zum Informations-Austausch«. Diese Zeichenverschlüsselungsart ist international als ISO-7-Bit-Code genormt, und es wäre wirklich nett, wenn alle sich daran halten würden. Tatsächlich aber finden wir zum Beispiel bei unserem C 64 eine Abart des Normcodes, den Commodore-ASCII-Code. Über die damit erzwungenen Umrechnungen können alle diejenigen Dramen erzählen, die zum erstenmal einen (Nicht-Commodore-)Drucker an ihr Gerät anschließen oder aber blauäugig in den Online-Betrieb mit anderen Computern eintreten wollten.

Sehen wir uns zunächst einmal den ASCII-Code an. Es handelt sich um einen 7-Bit-Code, das heißt 128 Zeichen können in nur 7 Bit untergebracht werden (0000 0000 bis 01111111). Das achte Bit dient bei manchen Operationen mit Computer-Peripherie als Paritäts-Bit. Bei dieser Gelegenheit soll auch gleich erklärt werden, was Parität in diesem Zusammenhang bedeutet. Werden Daten übertragen, muß immer

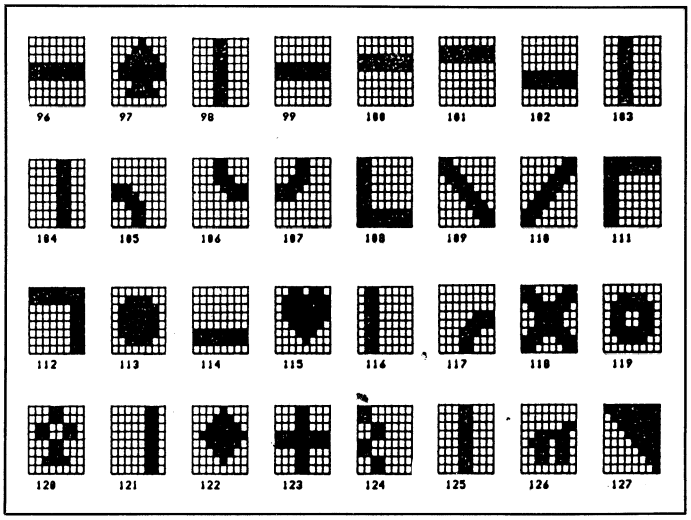


Bild 15. Grafikzeichen zu den entsprechenden CHR$-Codes

mit Übermittlungsfehlern gerechnet werden. Das Paritätsbit dient dazu festzustellen, ob ein Byte korrekt angekommen ist. Bei der sogenannten geraden Parität zählt man die Einser im Byte zusammen und setzt Bit 7 auf 1 wenn sich eine ungerade Zahl ergibt. Mit dem Paritätsbit haben wir dann eine gerade Zahl. Ist die Quersumme des Byte schon gerade, bleibt Bit 7 eine Null. Ebensogut kann man die ungerade Parität verwenden, indem dann Bit 7 so gewählt wird, daß sich immer eine ungerade Zahl ergibt. Welche Art der Parität zur Anwendung kommt, ist Vereinbarungssache. Nehmen wir mal an, es sei gerade Parität gefordert und ein Byte mit der Information 00010110 soll übermittelt werden. Die Quersumme ist 3, also ungerade. Das Paritätsbit muß auf 1 gesetzt werden. Wir senden das Byte 10010110. Der Empfänger überprüft zunächst auf gerade Parität und verwendet dann nur die Bits 0 bis 6. Doppelfehler, die mittels des Paritätsverfahrens nicht festgestellt werden können, sind sehr selten. Leider kann auf diese Weise nur bemerkt werden, daß ein Übertragungsfehler aufgetreten sein muß, aber nicht welcher. Die Information muß dann neu angefordert werden.

Sehen wir uns nun den Commodore-ASCII-Code an. Durch die Einbindung der Grafikzeichen brauchen wir mehr als die 128 Kombinationen. Commodore benutzt deswegen einen 8-Bit-Code. Mit dem Basic-Befehl CHR$(x) können Sie sich alle 256 Möglichkeiten ansehen. Erschwerend kommt aber noch hinzu, daß wir nicht nur einen Zeichensatz, sondern deren vier zur Verfügung haben, die durch den jeweiligen Schreibmodus ansprechbar sind (Klein-/Großschriftmodus, Großschriftmodus, beide Modi mit Reverse-ON oder OFF). Im Zeichen-ROM liegen insgesamt 512 Muster abrufbereit. Zu diesen kommen beim CHR$-Befehl noch eine ganze Reihe von Steuerzeichen hinzu... die Verwirrung ist perfekt! Wir wollen an dieser Stelle keine Entwirrung vornehmen, sondern wir durchschlagen den Gordischen Knoten, indem wir nur die ersten 128 Zeichen mit den ASCII-Zeichen vergleichen. In Bild 14 und 15 finden Sie unsere Gegenüberstellung.

Einige Kombinationen dienen als Steuer-Codes. (Die Bedeutung der dabei verwendeten Abkürzungen sehen Sie in Bild 16.)

Nur ein Teil dieser Codes wird tatsächlich genutzt. Andere haben – je nach Gerät an das sie gesandt werden – unterschiedliche Bedeutungen. Denken Sie dabei nur mal an die verschiedenen Betriebssysteme des Commodore-Druckers 1526, wo man bei dem einen mit CHR$(1), bei dem anderen mit CHR$(14) den Breitschrift-Modus anschaltet. Innerhalb unseres Computers werden offensichtlich bestimmte Codes anders genutzt. Das sind:

Anstelle von	geschieht folgendes:
ENQ	Zeichen weiß
BS	Blockieren der Umschaltung Klein-/Großschrift
HT	Zulassen der obigen Umschaltung
DC1	Cursor abwärts
DC2	Reverse-Modus an
DC3	Cursor in HOME-Position
DC4	INST/DEL
FS	Zeichen rot
GS	Cursor rechts
RS	Zeichen grün
US	Zeichen blau

Der auffälligste Unterschied ist der, daß beim Commodore-ASCII anstelle der Kleinbuchstaben Grafikzeichen liegen. Sollte anstelle des Normalmodus der Klein-/Großschriftmodus eingeschaltet sein, findet man anstelle der Großbuchstaben die kleinen.

Jetzt haben wir alle nötigen Kenntnisse, um die CHRGET-Routine in unserem Computer zu verstehen.

NUL	Null	
SOH	Start of heading	Beginn des Kopfes
STX	Start of text	Textbeginn
ETX	End of text	Textende
EOT	End of transmission	Übertragungsende
ENQ	Inquiry	Anfrage
ACK	Acknowledge	Bestätigung
BEL	Bell	Klingel
BS	Backspace	Zurücksetzen
HT	Horizontal tabul.	Horizontaltabulator
LF	Line feed	Zeilenvorschub
VT	Vertical tabulator	Vertikaltabulator
FF	Form feed	Formatvorschub
CR	Carriage return	Wagenrücklauf/ Zeilenwechsel
SO	Shift out	Rückschaltung
SI	Shift in	Dauerumschaltung
DLE	Data link escape	Datenverbindungs-umschaltung
DC1-4	Device control	Gerätesteuerung
NAK	Negative acknowl.	Negativ-Bestätigung
SYN	Synchronous idle	Synchronisations-Leerlauf
ETB	End of transmission block	Ende des Übertragungsblockes
CAN	Cancel	Annullieren
EM	End of medium	Datenträgerende
SUB	Substitute	Ersetzen
ESC	Escape	Umschaltung
FS	File separator	Dateitrennzeichen
GS	Group separator	Gruppentrennzeichen
RS	Record separator	Satztrennzeichen
US	Unit separator	Einheiten-Trennz.
SP	Space	Leerzeichen
DEL	Delete	Löschzeichen

Bild 16. Die Bedeutung der Abkürzungen im ASCII Code

25. Die CHRGET-Routine

Das Kürzel CHRGET kommt von »Get a character«, was bei uns heißt: »Hole ein Zeichen«. Es handelt sich um eine sehr häufig benutzte Routine unseres Basic-Interpreters, die – wie schon vorhin erwähnt – komplett in der Zeropage steht. Wenn Sie mit dem SMON mal nachsehen wollen, dann geben Sie den Befehl

D 0073 008B

ein. Sie haben dann die komplette Routine vor sich:

```
0073  E6  7A    INC  7A
0075  D0  02    BNE  0079
0077  E6  7B    INC  7B
0079  AD  2502  LDA  0225
007C  C9  3A    CMP  #3A
007E  B0  0A    BCS  008A
0080  C9  20    CMP  #20
0082  F0  EF    BEQ  0073
0084  38        SEC
0085  E9  30    SBC  #30
0087  38        SEC
0088  E9  D0    SBC  #D0
008A  60        RTS
```

Eventuell sieht die Zeile 0079 bei Ihnen anders aus. Das liegt dann an den Speicherstellen 7A und 7B, welche einen Zeiger darstellen (LSB=7A und MSB=7B), der bei Ihnen gerade auf einen anderen Platz zeigt als auf $0225.

Diese CHRGET-Routine besteht aus drei Teilen:

Zeilen 0073 bis 0079

Weiterstellen des CHRGET-Zeigers und Einladen des dadurch angezeigten Speicherzelleninhaltes in den Akku.

Zeilen 007C bis 0082

Prüfroutinen

Zeilen 0084 bis 008A

Flaggen-Routinen

Im ersten Teil haben wir schon gleich etwas neues vor uns: ein sich selbst veränderndes Programm. Die Speicherstelle (aus dem Basic-Eingabepuffer), aus der der Akku ein Zeichen holt, wird um 1 weitergezählt mit INC 7A.

Dabei handelt es sich um das LSB der Adresse und die nächste Zeile prüft, ob ein Überlauf (255+1) stattgefunden hat:

BNE 0079.

Diese Technik kennen wir schon aus den letzten Folgen: Bei Überlauf wird die Z-Flagge auf 1 gesetzt und der BNE-Befehl führt keinen Sprung herbei. Den Offset von 02 können wir leicht nachrechnen: Der Programmzähler steht schon auf 0077. Die Zieladresse 0079 ist also noch 2 Byte entfernt. Hat eine Überschreitung des Höchstwertes 255 stattgefunden, dann muß das dazugehörige MSB um 1 erhöht werden. Dies tut die nächste Zeile: INC 7B

In beiden Fällen ist nun der Zeiger 7A/7B um eine Stelle weitergerückt und der Inhalt der dadurch angezeigten Speicherstelle wird in den Akku geladen. Zwei Dinge können wir uns aus diesem kurzen Programmteil merken:

1) Wie man eine 16-Bit-Zahl hoch- (oder auch herunter-) zählt und

2) eine Möglichkeit, Zeiger einzusetzen. Wir werden noch eine Reihe anderer Zeigertypen kennenlernen und sehen, daß es nicht immer so direkt zugeht wie hier.

Im zweiten Teil finden wir die Prüfroutinen. Die Vergleichsbefehle beschränken sich auf den Akkuinhalt, also CMP.

CMP #3A testet, in welcher Beziehung das im Akku befindliche Zeichen zum Wert $3A = dezimal 58 steht. Erinnern wir uns an das Schema in Bild 14:

1) Commodore-ASCII-Code im Akku größer als 58, also Zeichen hinter dem Doppelpunkt (Buchstaben, Grafikzeichen, einige Sonderzeichen). Dann ist die Carry-Flagge = 1, N- und Z-Flagge sind 0.

2) Im Akku steht genau der Code 58, also der Doppelpunkt. Dann sind Carry-Bit und Z-Flagge = 1, nur die N-Flagge = 0.

3) Der Code des Zeichens im Akku ist kleiner als 58 (das wären alle Zahlen, einige Sonderzeichen und Steuerzeichen). In diesem Fall ist die N-Flagge = 1. Die beiden anderen Flaggen zeigen Null.

Der nun folgende Befehl BCS 008A überprüft die Carry-Flagge. Wenn sie gesetzt ist, wenn also der Code im Akku größer oder gleich dem eines Doppelpunktes (58) ist, springt der Programmzähler zum RTS. Der Code (und auch die Flaggen) wird unverändert zum aufrufenden Hauptprogramm weitergegeben. Zur Übung können Sie ja nochmal den Offset nachrechnen. Der Rest des Programms wird nur noch durchlaufen, wenn Codes kleiner als 58 im Akku stehen.

Die nächste Zeile CMP #20 dient zum Vergleich des Space-Codes $20 = dezimal 32 (Leertaste). Die Flaggen treten dann, wie schon oben beim ersten Vergleich gezeigt, je nach Akku-Inhalt auf. Durch die Verzweigung BEQ 0073 erfolgt ein Rücksprung zum Beginn der CHRGET-Routine dann, wenn die Z-Flagge gesetzt ist, also ein Space-Code im Akku liegt. Somit werden die Leerzeichen einfach übersprungen und das nächste Zeichen geholt. Alle anderen Zeichen, die bis hierher durchgehalten haben, werden nun im letzten Teil der CHRGET-Routine einer Prozedur unterworfen, die ich Flaggen-Routine genannt habe.

Durch zwei aufeinanderfolgende Subtraktionen, die insgesamt den Wert im Akku unverändert lassen (es wird 256 abgezogen), wird die Carry-Flagge beeinflußt. Verfolgen wir, was da passiert:

SEC dient als Vorbereitung für die folgende Subtraktion.

SBC #30 zieht vom Akku-Inhalt $30 = dezimal 48 ab. Wir wissen inzwischen, daß das der Addition des Zweierkomplementes entspricht. Dieses ist (rechnen Sie mal nach!) 1101 0000.

Nehmen wir mal an, wir hätten den Code der Zahl 4 (also dezimal 52 oder $34) im Akku stehen. Die Rechnung sieht dann so aus:

```
52      0011  0100
        1101  0000
      + ---------
    (1) 0000  0100
```

Das Ergebnis ist also 4, der Übertrag wird vernachlässigt.

Als anderes Beispiel sei nun der Code für das Ausrufungszeichen im Akku (dezimal 33 = $21 = binär 0010 0001). Die Rechnung ist dann:

```
33      0010  0001
+       1101  0000
        ---------
        1111  0001
```

Das Ergebnis ist —15.

Alle Codes, die nicht für Zahlen stehen, haben nach dieser Subtraktion ein negatives Ergebnis im Akku hinterlassen und durch das »Borgen« das Carry-Bit gelöscht.

Nun machen wir weiter ab Zeile 0087:

SEC

SBC #D0

Wir ziehen $D0 = dezimal 208 ab. Das Zweierkomplement ist: ...Doch da kommen wir ins Stocken! Denn dieses Zweierkomplement ist nicht mehr mit 8-Bit-Zahlen darzustellen. Schon die Zahl 208 im Binärformat (1101 0000) würde als negative Zahl angesehen werden, weil Bit 7 gleich 1 ist. Wir machen es uns einfach und sagen, daß sich das Zweierkomplement wie bisher bilden läßt, aber dabei das Carry-Bit mit einbezogen wird. Unser Zweierkomplement ist dann also: 0011 0000 und das Carry-Bit ist gelöscht. Nun nehmen wir unser erstes Beispiel. Dort war nach der Subtraktion im Akku eine 4 verblieben:

```
        0000  0100
+       0011  0000
        ---------
        0011  0100
```

Das ist wieder unser ursprünglicher Wert dezimal 52 = $34 = Code für die Zahl 4. Das Carry-Bit bleibt gelöscht.

Im zweiten Beispiel mit dem Ausrufungszeichen stand noch im Akku eine —15:

```
        1111  0001
+       0011  0000
        ---------
(1)     0010  0001
```

Da haben wir wieder den Code für das Ausrufungszeichen ($21 = dezimal 33) im Akku und ein gesetztes Carry-Bit. Was kommt also bei der CHRGET-Routine heraus?

1) Alle Zeichen außer dem Space werden unverändert an das aufrufende Programm über den Akku weitergegeben. Space wird unterdrückt.

2) Bei allen Zeichen außer bei den Zahlen ist das Carry-Bit gesetzt.

3) Manche der aufrufenden Routinen überprüfen außer dem Zustand der Carry-Flagge auch den der Z- oder N-Flagge, die ja beim ersten CMP-Befehl ebenfalls gesetzt werden. So liefert die CHRGET-Routine noch weitere Informationen.

In der einschlägigen Literatur stoßen Sie auch auf eine Routine, die CHRGOT genannt wird. Es handelt sich dabei ebenfalls um die hier beschriebene CHRGET-Routine, nur erfolgt der Einsprung nicht bei $0073, sondern bei $0079. Der Zeiger $007A/7B wird in diesem Fall nicht weitergestellt. Das vorher schon einmal in den Akku geladene Zeichen wird damit noch einmal angesprochen (got ist die Vergangenheitsform von get).

Mit dem CHRGET-Programm haben wir eines der wichtigsten Unterprogramme unserer computerinternen Software kennengelernt. Will man sich Interpreter-Routinen zunutze machen, stolpert man ständig darüber. Außerdem aber liegt die CHRGET-Routine im RAM. Das bedeutet, daß wir sie ohne weiteres für unsere Zwecke verändern können. Ein Beispiel für so eine Änderung hat Christoph Sauer in seiner Serie über den »gläsernen VC 20« in der Ausgabe 9 (Seite 158) gezeigt. Dort wird die CHRGET-Routine nach dem LDA angezapft und auf das Pi-Zeichen geprüft, das neuen Befehlen vorangestellt wurde. Sehen Sie sich das Programm dort (auf Seite 160f.) mal genau an, viel kann man durch Nachvollziehen fremder Programme für die eigene Programmiertechnik lernen.

26. Die indizierte Adressierung

Indizieren heißt, etwas mit einem Index, also einem Zeichen oder einer Nummer, zu versehen. Beispielsweise bezeichnet man in der Mathematik die beiden Lösungen einer quadratischen Gleichung häufig als X1 und X2. Dabei ist dann die Ziffer (1 oder 2) der Index und X ist eine indizierte Größe. Man geht also aus von einer festgelegten Grundmenge (Lösungsmenge X) und trifft durch den Index eine weitere Unterscheidung.

So ähnlich können wir uns auch die Funktion der indizierten Adressierung bei der Assembler-Programmierung vorstellen. Nehmen wir als Beispiel den Befehl

```
LDA 1500,X
```

Man spricht hier von einer absolut-X-indizierten Adressierung. Das Assemblerwort LDA ist uns bekannt: Lade den Akku. Woher soll der für den Akku bestimmte Inhalt geholt werden? Aus der Speicherzelle, die sich durch 1500 plus Inhalt des X-Registers ergibt. Steht also im X-Register zum Zeitpunkt des Befehlsaufrufes eine 5, dann wird der Akku aus Speicherzelle 1500+5, also 1505, geladen. Das X-Register kann Werte von 0 bis $FF (dez. 255) enthalten. Die Ähnlichkeit sieht also so aus:

Aus einer Gesamtmenge von 256 Adressen, die durch die Anfangsadresse (bei unserem Beispiel 1500) und die möglichen 256 Belegungen des X-Registers festgelegt sind (die Grundmenge), werden je nach X-Registerinhalt einzelne Adressen unterschieden und adressiert. Das X-Register fungiert dabei als ein Index, weswegen man auch oft die Bezeichnung »Index-Register X« in der Literatur findet.

Ebenfalls als Index-Register kann das Y-Register dienen, was zum Beispiel zum Befehl

```
LDX 1500,Y
```

führen kann. Dies ist dann eine absolut-Y-indizierte Adressierung.

Genauso wie man die normale absolute Adresse (also zum Beispiel 1500) als Basis der Indizierung durch das X- oder das Y-Register verwenden kann, ist das auch mit eine Zeropage-Adresse möglich. So gibt es zum Beispiel die Befehle

```
LDY 2B,X
```

oder

```
STX 19,Y
```

Man nennt diese Art der Adressierung dann Zeropage-absolut-X-indiziert beziehungsweise -Y-indiziert.

Weil die Zeropage aber nur 256 Adressen umfaßt, andererseits jedoch die Indexregister auch 256 Werte annehmen können, kann es geschehen (wenn man nicht aufpaßt), daß die Summe aus der Basisadresse (zum Beispiel $2B) und dem Indexregisterinhalt größer als 256 wird. Wenn zum Beispiel in dem Befehl

```
LDA FE,X
```

der X-Registerinhalt 2 beträgt, ergäbe sich $FE+$02=$0100. In diesem Fall wird aber nicht der Inhalt von $0100 in den Akku geladen, sondern der Befehl spricht die Speicherstelle $00 an. Der Grund dafür liegt in der Tatsache, daß unser Prozessor den Befehl als 2-Byte-Befehl interpretiert – das 2. Byte ist die Zeropageadresse, die sich als Summe ergibt – und deswegen nur das LSB der Adresse beachtet. Von $0100 ist das LSB aber $00. Mit anderen Worten: Die Zeropage-absolut-indizierten Befehle lassen einen Zugriff nur auf die Zeropage selbst zu. Dieses Verhalten muß man beim Programmieren beachten.

Wir wollen nochmal zusammenfassen. Vier neue Adressierungsarten haben wir kennengelernt:

Befehl	Indizierte Adressierung			
	absolut		Null-Seite-absolut	
	X	Y	X	Y
LDA	+	+	+	-
LDX	-	+	-	+
LDY	+	-	+	-
STA	+	+	+	-
STX	-	-	-	+
STY	-	-	+	-
RTS	/	/	/	/
INX	/	/	/	/
INY	/	/	/	/
INC	+	-	+	-
DEX	/	/	/	/
DEY	/	/	/	/
DEC	+	-	+	-
SED	/	/	/	/
CLD	/	/	/	/
BNE	/	/	/	/
ADC	+	+	+	-
CLC	/	/	/	/
SBC	+	+	+	-
SEC	/	/	/	/
BEQ	/	/	/	/
BCC	/	/	/	/
BCS	/	/	/	/
BMI	/	/	/	/
BPL	/	/	/	/
BVC	/	/	/	/
BVS	/	/	/	/
CMP	+	+	+	-
CPX	-	-	-	-
CPY	-	-	-	-
BIT	-	-	-	-
CLV	/	/	/	/
NOP	/	/	/	/
TAX	/	/	/	/
TAY	/	/	/	/
TXA	/	/	/	/
TYA	/	/	/	/
JMP	-	-	-	-
JSR	-	-	-	-

\+ anwendbar
\- nicht erlaubt
/ weder absolute noch Zeropage-Adressierung möglich

Tabelle 5. Anwendbarkeit der indizierten Adressierungsarten auf die bisher gelernten Assembler-Befehle.

Absolut-X-indiziert zum Beispiel LDA 1500,X
Absolut-Y-indiziert zum Beispiel LDA 1500,Y
Zero-page-absolut-X-indiziert zum Beispiel LDA 2B,X
Zero-page-absolut-Y-indiziert zum Beispiel LDX 2B,Y

Die Verwendung des Y-Registers als Indexregister ist stark eingeschränkt. Nur bei wenigen Befehlen ist sie erlaubt (tatsächlich nur LDX und STX bei Zero-page-absolut-indizierter Adressierung). In der Tabelle 5 sehen Sie, welche bisher behandelten Befehle wie mit der indizierten Adressierung verwendet werden dürfen.

Es gibt noch zwei weitere Arten einer indizierten Adressierung, auf die wir noch zu sprechen kommen werden.

27. Einige Nachzügler: Die Befehle BIT, CLV, NOP und TAX, TAY, TXA, TYA

Wir wollen noch ein bißchen aufräumen: Ein paar Befehle, die bisher zu keinem Gebiet so richtig paßten, sollen jetzt behandelt werden.

BIT: Dieser Befehl heißt »Bit-Test» und paßt von daher eigentlich zu den in Kap. 23 behandelten Vergleichsbefehlen. Die Behandlung der Flaggen ist aber völlig anders. Nehmen wir das Beispiel

```
BIT 1500
```

Folgendes passiert: Der Inhalt der Speicherstelle $1500 wird mit dem Inhalt des Akku UND-verknüpft, das Ergebnis in der Z-Flagge angezeigt und Bit 7 sowie Bit 6 von $1500 in die N- beziehungsweise die V-Flagge übertragen. Weder Akku noch Inhalt von $1500 verändern sich dabei.

Das ging ein bißchen holterdipolter. Sehen wir uns das jetzt mal ganz langsam Schritt für Schritt an! Zunächst die UND-Verknüpfung. Bit für Bit wird der Akku-Inhalt mit dem Inhalt der adressierten Speicherstelle UND-verknüpft. Dabei gelten folgende Regeln (die Leser der Grafik-Serie kennen das ja schon):

```
0 UND 0 = 0
0 UND 1 = 0
1 UND 0 = 0
1 UND 1 = 1
```

Nur dann also, wenn die entsprechenden Bits im Akku und in 1500 gleich 1 sind, ergibt sich bei der UND-Verknüpfung eine 1. Man stellt sowas meist in einer sogenannten Wahrheitstabelle zusammen (Tabelle 6).

UND	0	1
0	0	0
1	0	1

Tabelle 6. Wahrheitstabelle der logischen Verknüpfung UND

Nehmen wir als Beispiel mal an, im Akku stünde $0A und in der Speicherstelle $1500 wäre $09 enthalten. Die UND-Verknüpfung sieht dann so aus:

```
Akku   $0A  0000 1010
1500   $09  0000 1001
UND   ____________
            0000 1000
```

Das Ergebnis ist also $08. In der Z-Flagge wird in dem Fall, daß das Ergebnis der UND-Verknüpfung ungleich Null ist (wie hier) eine Null angezeigt, sonst eine 1.

Wir haben in unserem Zahlenbeispiel mit dem BIT-Befehl überprüft, ob die Bits 1 und 3 in Speicherstelle $1500 gelöscht sind. Dazu haben wir in den Akku eine sogenannte Maske (hier also $0A) geladen. Das Ergebnis sagt uns, daß nicht beide Bits gelöscht waren. Wäre der Inhalt von $1500 beispielsweise $10 gewesen (0001 0010), hätten wir in der Z-Flagge eine 1 gefunden. Daher der Name »Bit-Test«: Durch geeignete Maskenwahl kann praktisch jedes Bit überprüft werden. Dabei werden weder der Akku-Inhalt noch der Inhalt der angesprochenen Speicherstelle verändert.

Der BIT-Befehl hat aber noch mehr Auswirkungen: Die Bits 6 und 7 der geprüften Speicherzelle findet man nach Befehlsausführung in zwei Flaggen nochmal:

```
Bit 7 in der N-Flagge
Bit 6 in der V-Flagge
```

Damit kann man beispielsweise überprüfen, ob sich am adressierten Ort eine negative Zahl befindet. Alle drei Flaggen können ja nun mit den Branch-Befehlen abgefragt werden. Sie erkennen sicherlich schon, wie vielseitig dieser merkwürdige BIT-Befehl einsetzbar ist.

Adressierbar ist BIT entweder absolut (wie im obigen Beispiel) oder Zeropage-absolut. Je nachdem liegt er dann als 3-Byte- oder als 2-Byte-Befehl vor.

CLV: Dieser Befehl heißt »Clear oVerflow-flag«, also »lösche die Überlauf-Flagge«. Die V-Flagge war –wie Sie sich erinnern werden –unsere rote Ampel bei Rechenoperationen (siehe Kap. 16). Es ist ein 1-Byte-Befehl mit impliziter Adressierung und interessant daran ist, daß es keinen Befehl gibt, der das Gegenteil –also das Setzen der V-Flagge –bewirkt.

NOP: NOP steht für »No OPeration«, was bedeutet »keine Tätigkeit«. Das ist der Nichtstu-Befehl. Er tut aber doch etwas: Er sorgt dafür, daß der Befehlszähler weitergezählt wird und bewirkt eine Verzögerung von 2 Taktzyklen. NOP ist ein 1-Byte-Befehl mit impliziter Adressierung. Er wird in fertigen Programmen nur selten verwendet: Zur Erzeugung einer kurzen definierten Verzögerung. Meist gebraucht man ihn bei der Erstellung eines Programmes als Platzhalter oder bei der Fehlersuche, um zum Beispiel unerwünschte Sprünge zu ersetzen.

Die Transporteure: TAX, TAY, TXA und TYA

Ab und zu ist es nötig, Registerinhalte untereinander auszutauschen. Viele Dinge (Addition, Subtraktion und so weiter) können nur im Akku geschehen. Wenn wir eine solche Operation beispielsweise mit dem Inhalt des X-Registers ausführen wollen, verschieben wir diesen Inhalt mit dem Befehl TXA. »Transfer X into Accumulator« also »übertrage X-Register in den Akku« bedeutet das. Analog verwendet man TYA, um Y-Register-Inhalte in den Akku zu schieben oder für den umgekehrten Weg TAY beziehungsweise TAX (Akkuinhalt ins Y- beziehungsweise ins X-Register schieben). Genau genommen wird nicht übertragen, sondern nur kopiert: Die Register, aus denen verschoben wird, bleiben unverändert. Weil die jeweiligen Zielorte der Verschiebung (Akku, X- oder Y-Register) vom neuen Inhalt überschrieben werden, können sich auch Flaggen ändern. Betroffen sind von dieser Möglichkeit die N- und die Z-Flagge. Alle vier Befehle bestehen aus einem Byte und können natürlich nur implizit adressiert werden.

28. So springen die Assembler-Alchimisten: JMP, JSR

JMP und JSR entsprechen ungefähr den vom Basic her bekannten Befehlen GOTO und GOSUB.

JMP kommt von »JuMP to address«, also »springe zur angegebenen Adresse«. Nehmen wir uns wieder ein Beispiel vor:

```
JMP    1500
```

bewirkt einen Sprung zur Adresse 1500. Das funktioniert so: In den Programmzähler werden LSB und MSB der Ziel-

adresse geladen. Das war dann auch schon der Sprung, denn der Programmzähler ist der Pfadfinder des Computers: Die Adresse, die dort steht, wird als nächste bearbeitet. Schalten Sie doch mal den SMON ein (oder einen anderen Monitor) und sehen Sie sich das mit folgenden Befehlen an:

```
1400   JMP    1500
```

Dort unterbrechen wir den Computer mit

```
1500   BRK
```

So weit, so gut: Wir starten mit dem SMON-Kommando G 1400 und erhalten eine Registeranzeige mit dem Programmzählerstand 1501. Genau das hatten wir ja erwartet.
Weniger durchschaubar ist das folgende Beispiel:

```
1400   LDA    #00
1402   LDX    #16
1404   STA    1300
1407   STX    1301
140A   JMP    (1300)
```

Dazu gehört dann noch die Programmzeile:

```
1600   BRK
```

Wenn Sie das genauso eingegeben haben und dann mittels G 1400 starten, erhalten Sie eine Registeranzeige mit dem Programmzählerstand 1601.

Schon an der neuen Schreibweise des Argumentes in Zeile 140A werden Sie bemerkt haben, daß hier nicht mehr die normale absolute Adressierung wie zuvor angewendet wird. Dies ist eine neue Form: Die **indirekte Adressierung**. Indirekt deswegen, weil wir nicht mehr direkt die Zieladresse angeben, sondern einen sogenannten Vektor. Ein Vektor besteht aus zwei aufeinander folgenden Speicherzellen (hier also 1300 und 1301), die in der Form LSB/MSB die eigentliche Zieladresse enthalten. Das LSB von $1600 ist $00. Das haben wir über den Akku nach $1300 geladen. Das MSB $16 kam durch das X-Register an seinen Platz $1301:

```
Zieladresse  16     00
             MSB    LSB
             ↑      ↑
Vektor       1301   1300
```

Das ist die Methode der toten Briefkästen, die in Kreisen der Assembler-Alchimisten anscheinend genauso beliebt ist wie bei Agenten. So wie diese im hohlen Baum die Treffpunktanschrift hinterlegt finden, verläßt sich unser Computer auf die Speicherstellen 1300 und 1301 für die Angabe der Zieladresse.

Diese Art der Adressierung ist im wahrsten Sinn des Wortes ein Unikum: Es gibt sie nämlich nur für den JMP-Befehl! Davon wird allerdings dann auch recht häufig Gebrauch gemacht, zum Beispiel im Betriebssystem unseres Computers. Aber darüber und über die Vektoren, die dazu verwendet werden, soll ein andermal berichtet werden.

Wir dürfen nämlich nicht den anderen Sprungbefehl JSR vergessen. JSR steht für »Jump to SubRoutine«, was eingedeutscht etwa bedeutet »springe zum Unterprogramm«. Genauso wie in Basic Unterprogramme durch GOSUB Zeilennummer aufgerufen werden, kann das auch hier geschehen durch JSR Adresse. Hier ist nur die absolute Adressierung möglich. Das erste Beispiel soll uns zeigen, wie dieser Befehl funktioniert:

```
1400   JSR    1500
```

Dort soll dann erstmal stehen:

```
1500   BRK
```

Noch nicht starten!! Zunächst einmal verzeihen Sie mir diese Programmierer-Todsünde: Aus einem Unterprogramm heraus den Programmablauf zu beenden! Ich werd's auch nie wieder tun. Hier geschieht das nur zu Lehrzwecken. Was läuft ab: Der Programmzählerinhalt plus 2 wird auf den Stapel gelegt und dann die Adressse 1500 in den Programmzähler geladen. Ebenso kurz wie unklar! Was ist denn ein Stapel? Also langsam, Schritt für Schritt.

Der Sinn von Unterprogrammen ist ja, daß der Computer nach Ende der Bearbeitung wieder ins aufrufende Hauptprogramm zurückkehrt. Er muß sich aber dazu irgendwo merken, von wo aus er zum Unterprogramm gesprungen ist. Dazu verwendet er den Stapel. Das ist ein Speicherbereich ($0100 bis $01FF), der direkt vom Prozessor aus verwaltet wird. Die genaue Architektur und Handhabung dieses »Prozessor-Stack« werden wir noch in einer späteren Folge kennenlernen. Uns soll hier nur interessieren, daß es einen Zeiger gibt, der auf den nächsten freien Platz im Stapel weist und daß dieser Speicher von oben nach unten gefüllt wird (wie in Basic bei den Strings). Wenn Sie mit Hilfe des SMON mal in den Stapel hineinsehen wollen, dann geben Sie doch mal ein M 0100 01FF. Was nun genau bei Ihnen drin steht, ist sehr von der vorherigen Nutzung Ihres Computers abhängig. Der Mikroprozessor nutzt den Stapel bei sehr vielen Tätigkeiten. Es kommt auch nur auf den Teil des Stapels an, der durch den Stapelzeiger als gefüllt bezeichnet wird. Der Stapelzeiger wird beim SMON in der Registeranzeige als SP angezeigt. Wenn Ihr Stapelzeiger (prüfen Sie das doch mal durch Eingabe von R) nun zum Beispiel F6 zeigt, dann bedeutet das, daß alle Stapelplätze von $01F6 an abwärts frei und die oberhalb bis $01FF besetzt sind. Beim Nachsehen mit M 01F0 01FF finden Sie dann beispielsweise:

```
:01F0   20  00  20  AA  C1  FA  C0  46
:01F8   E1  E9  A7  A7  79  A6  9C  E3
```

Die Speicherstelle, auf die der Stapelzeiger weist, ist unterstrichen. Nun starten wir mit G 1400 unser kleines verbotenes Testprogramm. Es meldet sich die Registeranzeige. Im Stapelzeiger steht jetzt F4 (oder eben Ihr vorhergegangener SP minus 2). Wenn wir nun wieder im Stapel nachsehen mit M 01FO 01FF, dann finden wir im Gegensatz zur obigen Anzeige nun:

```
:01F0   20  AA  C1  FA  C0  02  14  46
                        --  ↑↑  ↑↑
:01F8   E1  E9  A7  A7  79  A6  9C  E3
```

Unterstrichen ist wieder das Ziel des Stapelzeigers, der jetzt zwei Plätze weitergerückt ist, um der durch Pfeile gekennzeichneten Adresse 1402 (als LSB/MSB) Raum zu schaffen. $1402 ist das letzte Byte des JSR-Befehls. Wie wir den Programmzähler kennen, ist er im allgemeinen immer einen Schritt voraus. Hier liegt er aber einen zurück, falls er nach Beendigung des Unterprogrammes an der notierten Adresse weitermacht. Dazu kommen wir gleich noch. Was wir am Programmzähler aber auch noch nach Ablauf unseres kurzen Beispielprogrammes ablesen können, ist die Tatsache, daß die Sprungadresse 1500 in ihn geschrieben wird, somit der Sprung dann also stattgefunden hat.
Nun bauen wir das kleine Programm etwas um:

```
1400   JSR    1500
1403   BRK
```

Das Unterprogramm soll nur aus dem Rücksprung bestehen:

```
1500   RTS
```

Verlangen Sie nun noch vor dem Start eine Registeranzeige mit R und merken Sie sich den Wert des Stapelzeigers. Dann starten Sie das Programm mit G 1400 und achten Sie auf die neue Registeranzeige. Zwei Dinge interessieren uns:
1) Der Wert des Stapelzeigers ist unverändert geblieben.
2) Der Programmzähler weist nun auf $1404.

Wenn Sie nun nochmal mit dem M-Befehl des SMON in den Stapel sehen, werden Sie unter Umständen zwar noch die Adresse 1402 dort finden (dann nämlich, wenn wir den Stapel seit dem letzten Programm nicht verändert haben). Wie Sie aber inzwischen wissen, hätte durch den neuen JSR-Befehl nochmal 1402 dort eingetragen sein müssen. Das stand da auch einige Mikrosekunden lang... bis der RTS-Befehl wirksam wurde. RTS macht ziemlich viel:
1) RTS holt die auf dem Stapel gespeicherte Adresse ab, und schreibt sie in den Programmzähler.

2) RTS vermindert dabei den Stapelzeiger um 2.
3) RTS addiert zum Programmzähler eine 1.

Deswegen kann das Programm also bei $1403 weiterlaufen und der Programmzähler nun hinter dem BRK-Befehl stehen.

Machen Sie doch mal etwas anscheinend total Verrücktes: Starten Sie mit G 1500. Es gibt da zwei Möglichkeiten, was geschehen kann: Entweder stand da noch vom ersten unterbrochenen Testprogramm die Adresse 1402. Dann endete nun alles mit einer Registeranzeige, bei der der Stapelzeiger um 2 höher gerutscht ist.

Oder da stand diese Adresse nicht mehr. Dann befinden Sie sich nun wieder im Basic. Wieso eigentlich? Als nächste Adresse finden Sie auf dem Stapel $E146 (dez.57670). Diese Adresse + 1 wird ja durch RTS in den Programmzähler gerufen. Ein Sprung an diese Adresse ist ein Sprung in ein Programm des Betriebssystems. Haben Sie ein ROM-Listing? Dann sehen Sie mal nach: Dort steht der Befehl...RTS. Dies neuerliche RTS holt nun jedenfalls die nächste Adresse vom Stapel: $A7E9 (dez.42985). Diese Adresse + 1 im Programmzähler führt unseren Computer in die Basic-Interpreter-Schleife, also ins Basic zurück.

Wir haben so viel über den Stapel gehört, daß wir JSR fast schon wieder aus den Augen verloren haben. Deswegen nochmal eine kurze Übersicht:
a) JSR speichert den Programmzählerwert des letzten Bytes des Befehls auf dem Stapel zum Beispiel 1402,
b) stellt dabei den Stapelzähler um 2 zurück zum Beispiel von $F6 nach $F4
c) schreibt in den Programmzähler die angegebene Zieladresse, zum Beispiel 1500
d) Das Unterprogramm wird abgearbeitet bis der RTS-Befehl auftaucht.
e) Dann wird die gemerkte Adresse +1 in den Programmzähler geschrieben, zum Beispiel 1402+1=1403
f) und dabei der Stapelzähler wieder um 2 erhöht, zum Beispiel von $F4 wieder zu $F6
g) Das Programm läuft nun wieder nach dem JSR-Befehl weiter, zum Beispiel also bei 1403.

Nun sollte eigentlich auch klar sein, warum ein Aussprung aus einem Unterprogramm oder ein Abbruch im Unterprogramm eine Programmierer-Todsünde ist: Der Stapelzeiger wird nicht zurückgestellt. Die gemerkte Rücksprungadresse versauert allmählich auf dem Stapel. Noch schlimmer sind solche Sachen in einer Schleife, wo mehrfach aus dem Unterprogramm ausgebrochen wird: Hier ist der Stapel bald voll Müll und der Computer beendet seine Zusammenarbeit mit dem Programmierer. Weil aber Basic-Programme nichts anderes sind als eine Folge von Maschinenprogrammen, die je nach Befehl durch den Interpreter aneinandergereiht werden, ist das auch in Basic eine Todsünde. Wir wollen aber nicht so hart mit uns umgehen: Wenn wir gelernt haben, wie man mit speziellen Assembler-Befehlen im Stapel herumschaufeln kann, dann haben wir bei richtiger Anwendung von vorneherein jedenfalls in diesem Punkt die Absolution erhalten.

29. Alles fließt: Fließkommazahlen

Jeder, der tiefer in die Geheimnisse der Assembler-Alchimie eindringen will, muß sich vertraut machen mit der häufigsten Art der Zahlenverarbeitung in unserem Computer. Das ist die Handhabung von Fließkommazahlen (auch Gleitkommazahlen genannt). Wir werden dazu folgende Fragen zu klären haben:
1) Was sind Fließkommazahlen?
2) Wie sehen sie im binären Zahlensystem aus?
3) Wie behandelt unser Computer positive und negative Fließkommazahlen?
4) Wie können wir als Programmierer Einfluß nehmen auf die Verarbeitung dieser Zahlen im Computer?

Die Behandlung dieser vier Fragen wird uns eine ganze Weile beschäftigen. Fangen wir mit der ersten an: In Standardwerken der Mathematik werden Sie lange suchen müssen, um den Begriff »Fließkommazahl« zu finden. Im deutschen Sprachraum gibt es häufiger die Bezeichnung »wissenschaftliche Zahlendarstellung«. Das klingt sehr hochgestochen und ist eigentlich ganz einfach. Leser der Grafik-Serie werden sich vielleicht noch erinnern: Die Zahl 1000 kann man auf verschiedene Weise darstellen:

1000 = 10 * 10 * 10 = 10^3 (in Basic 10↑3)

Die hochgestellte Zahl (in Computerschreibweise: Die Zahl hinter dem Hochpfeil) ist hier gleich der Anzahl der Stellen minus 1 (1000 hat vier Stellen, also ist die Hochzahl eine 3). Diese Hochzahl nennt man Exponent (vom lateinischen exponere = anzeigen, herausheben). Nehmen wir nun einige andere Zahlen:

200 = 2 * 100 = 2 * 10↑2

oder

2500 = 2,5 * 1000 = 2,5 * 10↑3

Ich glaube, jetzt beginnt es Ihnen klarzuwerden, daß man auf diese Art wohl alle Zahlen irgendwie darstellen kann. Man dröselt die Zahlen auseinander, bildet ein Produkt, von dem der eine Multiplikator durch 10 teilbar ist (durch die Basis unseres normalen Zahlensystems). Genauer gesagt: Ein Faktor (also in den Beispielen 1000 oder 100) ist darstellbar als Potenz von 10. Der andere Faktor (in den Beispielen 1 oder 2 oder 2,5) wird Mantisse (vom lateinischen manitissa = Zugabe, Anhang, Schleppe) genannt. Sehen wir uns nochmal 2500 an:

```
2500 =   2,5 * 1000 =   2,5 * 10↑3
     =    25 *  100 =    25 * 10↑2
     =   250 *   10 =   250 * 10↑1
     =  2500 *    1 =  2500 * 10↑0
```

Das letzte war nur der Vollständigkeit halber, denn irgendeine Zahl hoch 0 ist immer 1. Man kann auch aus der 2500 folgendes machen:

2500 = 0,25 * 10000 = 0,25 * 10↑4
oder = 0,025 * 100000 = 0,025 * 10↑5

und so weiter. Oder anders herum:

2500 = 25000 * 0,1 = 25000 * 10↑-1
= 250000 * 0,01 = 250000 * 10↑-2

und so weiter.

Dabei bedeutet:

$10^{-2} = 1/10^{-2} = 0,01$

Man kann sich das merken, indem man die Anzahl der Stellen zählt, um die man das Komma verschiebt. Diese Anzahl addiert man dann zur Hochzahl. Zur Erläuterung:

$0,12345 = 1,2345 * 10^{-1}$

Wir haben das Komma um eine Stelle nach rechts gerückt, weshalb wir die Hochzahl -1 schreiben müssen (vorher war da nämlich unsichtbar die Hochzahl 0: und 10↑0=1).

$0,12345 = 123,45 * 10^{-3}$

Hier wurde das Komma um drei Stellen nach rechts verschoben. Daher der Exponent -3. Sie sehen folgenden Zusammenhang:

Komma eine Stelle nach rechts verschoben: Exponent + (-1).
Zum Beispiel
$0,1234*10^{-2} = 1,234*10^{-3}$

Komma eine Stelle nach links verschoben: Exponent +1.
Zum Beispiel
$3,14*10^{-2} = 0,314*10^{-3}$

Verstehen Sie nun, warum man diese Art der Zahlendarstellung Fließkomma- oder Gleitkommazahlen nennt?

Vielleicht sehen Sie aber noch nicht den Sinn der Fließkommazahlen ein. Dazu gebe ich Ihnen zwei einsichtige Bei-

spiele. Der Atomkern eines Heliumatoms wiegt etwa (halten Sie sich fest):
0,000 000 000 000 000 000 000 000 006 643 kg.

Sehr unbequem, diese ganzen Nullen immer mitzuschleppen. Wir verschieben deshalb das Komma um 27 Stellen nach rechts und schreiben dann

$6{,}643 * 10^{-27}$ kg.

2. Beispiel: Wir haben einen Ballon mit diesem Gas gefüllt. Bei normalen Temperatur- und Luftdruckbedingungen befinden sich in einem Kubikzentimeter im Ballon ungefähr (nochmal festhalten!):
26 900 000 000 000 000 000 Heliumatome

Wieder eine recht unangenehme Nullschlepperei. Wir verschieben das Komma um 19 Stellen nach links und erhalten $2{,}69 * 10^{-19}$ Heliumatome. Fein, nicht wahr!

Abgesehen von der höheren Bequemlichkeit: Der Computer müßte allerhand Speicherplatz zur Handhabung der vielen Nullen bereitstellen. Mit BCD-Zahlen könnten wir zwar jede Zahl erfassen, hätten aber immer unterschiedlich viele Bytes zu verarbeiten. Wenn wir Fließkommazahlen verwenden, können wir – wie Sie noch sehen werden – jede (na sagen wir mal: fast jede) Zahl in der gleichen Anzahl Bytes aufbewahren.

Vom Basic her kennen Sie Fließkommazahlen auch (hier wird das Komma allerdings durch den Punkt ersetzt, entsprechend der angloamerikanischen Schreibweise). Das sind die, wo man zum Beispiel schreibt 6.02E23 oder 6.02E+23, was dann bedeutet $6{,}02 * 10^{-23}$. E steht dort für Zehnerexponent. Durch die Art, wie Fließkommazahlen im normalen Computerdasein gespeichert werden, ergeben sich obere und untere Grenzen. Die höchste in Basic verarbeitbare Zahl im C 64 ist

$+1.70141183 * 10^{-38}$

Größere Zahlen verursachen in Basic einen OVERFLOW ERROR. Was in Maschinensprache mit größeren Zahlen geschieht, ist weitgehend unsere Sache. Die dem Betrag nach kleinste verarbeitbare Zahl ist

$\pm\ 2.93873588 * 10^{-39}$

In Basic arbeitet bei Unterschreitung der Computer einfach mit einer Null weiter. Für die Behandlung in Maschinensprache sind ebenfalls wir als Programmierer verantwortlich.

Für diesmal sei's genug der Zahlenspiele: Später werden wir uns weiter mit Fließkommazahlen befassen.

30. Die USR-Funktion

Wieder einmal soll uns das Zusammenspiel von Basic und Maschinensprache beschäftigen. Einen Aufruf von Maschinenroutinen – nämlich den mit SYS – haben wir schon kennengelernt. Wir POKEten die zu übergebenden Werte an die Abrufspeicherstellen. Bei diesen Werten hat es sich um einfache Integerzahlen gehandelt, zum Beispiel die Anzahl der Glieder einer zu summierenden arithmetischen Reihe. Was tun wir aber, wenn wir Fließkommavariable an ein Maschinenprogramm übermitteln wollen? Gewiß, werden Sie sagen, lernen wir das ja noch und können dann entsprechende POKE-Kommandos geben. Damit haben Sie auch recht, nur ist das dann der »harte« Weg. Es gibt auch einen problemlosen »weichen« Weg, nämlich das USR-Kommando.

USR ist ein Basic-Befehl und rührt her von »USeR callable machine language subroutine«, also »durch den Benutzer aufrufbares Maschinensprachunterprogramm«. Darin liegt eigentlich noch nichts Neues gegenüber dem SYS-Befehl. Im Gegensatz zu SYS – wo das Argument die Einsprungadresse des Maschinenprogrammes ist – übergibt USR als Argument eine beliebige Fließkommavariable in festgelegter Form an eine sehr nützliche Speicherstellenkombination, den Fließkomma-Akkumulator 1, von uns künftig einfach FAC genannt. Der FAC belegt die Speicherstellen 97 bis 102 ($61 bis $66). Wenn das eventuell in Basic benötigte Ergebnis dort auch in der vorgeschriebenen Form abgelegt wird, kann es im Basic-Programm weiterverwendet werden. Keine Angst, dazu kommen wir bei der weiteren Behandlung der Fließkommazahlen noch ganz ausführlich zu sprechen. Heute soll uns das noch nicht belasten. Als Argument kann man nämlich auch irgendeine bedeutungslose Größe, ein sogenanntes Dummy angeben, das dann gar nicht weiter verwendet wird. Der USR-Befehl dient in diesem Fall lediglich dem bequemen Ansteuern eines Maschinenprogrammes.

Woher weiß unser Computer beim USR-Befehl, welche Maschinenroutine er im 64-KByte-Speicher bearbeiten soll? Beim SYS-Befehl ist das klar: Das Argument sagt es:

```
SYS 24345
```

läßt den Programmzähler auf dez.24345 zeigen. Aber wenn wir eingeben:

```
USR(24345)
```

dann packt der Computer die Zahl 24345 als Fließkommavariable in den FAC und meldet dann einen SYNTAX ERROR. Das liegt daran, daß der Basic-Interpreter beim USR-Befehl einen der oben kennengelernten indirekten Sprünge vollführt:

```
JMP     (311)
```

$311/312 (in dezimal 785/786) ist also ein Vektor, und der weist im Normalfall zu einer Routine, die den SYNTAX ERROR ausgibt (dez. 45640). Bevor wir also den USR-Befehl geben, müssen wir in diesen Vektor die Startadresse unserer Maschinenroutine schreiben:

dez. 24345 = $5F19

LSB $19 = dez. 25 in Speicher 785 mit
POKE 785,25

MSB $5F = dez. 95 in Speicher 786 mit
POKE 786,95

Jetzt weiß der Computer, wohin er beim USR-Aufruf springen soll, und solange, bis wir den Vektor wieder ändern, führt er bei jedem USR-Befehl unser bei 24345 stehendes Maschinenprogramm aus. Wir müssen nur noch dafür sorgen, daß dort dann auch wirklich eines anfängt. Ein Beispiel werden wir nachher noch behandeln.

31. Der harte Kern: Nochmal Speicherfragen

Die Struktur des C 64-Speichers ist vereinfacht schon in der Grafik-Serie und zu Beginn dieses Kurses gezeigt worden. Dabei tauchten zwei ROM-Bereiche auf, die wir Basic-Interpreter und Betriebssystem genannt haben. Diese Unterteilung ist nicht ganz korrekt. Wenn Sie über ein ROM-Listing verfügen und beispielsweise das Ende des ROM-Bereiches von $A000 bis $BFFF sowie den Anfang des oberen ROM ($E000 bis $FFFF) untersuchen, dann stellen Sie fest, daß ab dez. 49087 ($BFBF) die Basic-Funktion EXP bearbeitet wird. Der letzte Befehl vor $C000 beendet diese Funktion aber nicht etwa, sondern dort steht:

```
JMP     E000
```

Tatsächlich läuft ab $E000 bis $E042 die Bearbeitung der EXP-Funktion munter weiter, und auch danach finden sich allerlei Basic-Befehle (SIN, COS und so weiter). Da liegt also keine klare Trennung vor, sondern ein Mischmasch. Wir sollten uns vielleicht angewöhnen – statt vom Interpreter und dem Betriebssystem –, vom unteren und oberen ROM-Bereich zu sprechen.

Eine andere Unterscheidung ist dagegen sinnvoll: Wie einige Besitzer neuerer Commodore 64 sicherlich bemerkt

haben, sind Teile der ROM-Routinen im Laufe der Zeit verändert worden. Hauptsächlich geht es bei den aktuellen Neuerungen dieser internen Maschinenprogramme um die Farbgebung der Zeichen. Man kann eigentlich nie so recht wissen, was den Software-Planern von Commodore noch alles einfällt. Jedenfalls können deren Ideen manchmal recht dramatische Folgen haben, nämlich dann, wenn Sie ein fabelhaftes Maschinenprogramm gebaut haben, welches ROM-Routinen direkt verwendet. Der Programmierer spielt auf diese Weise eine milde Form des russischen Roulettes. Glücklicherweise halten sich die Änderungen in Grenzen, und wir dokumentieren unsere Programme ja auch immer gut (Sie etwa nicht??). Notwendige Umbauten können also leicht vonstatten gehen.

Ganz ohne ROM-Routinen-Verwendung kommt man eigentlich kaum aus. Es gibt aber einen ROM-Bereich, für den Commodore verspricht, keinerlei Änderungen durchzuführen: die Kernel-Sprungtabelle.

Das ist ein Programmbereich ($FF81 bis $FFF5), in dem 39 JMP-Befehle enthalten sind (zum Teil in absoluter, aber auch in indirekter Adressierung). Jeder dieser Sprungbefehle weist auf die Einsprungadresse eines Maschinenprogrammes. Da finden sich alle wichtigen Ein/Ausgabe-Operationen, Systemtakt- und Uhrsteuerungen und anderes mehr. Wir werden uns nach und nach damit vertraut machen. In der Tabelle 7 sind die Kernel-Adressen und ihre Funktion aufgeführt. Manche davon können ohne jede Vorbereitung benutzt werden, andere brauchen bestimmte Routinen oder Angaben, um sinnvoll zu arbeiten.

Die Absicht von Commodore ist es, daß jeder Aufruf von zum Beispiel $FFD2 die Ausgabe eines Zeichens bewirkt, und zwar unabhängig davon, welchen Computer in welcher Version wir benutzen. Das Programm, welches diese Zeichenausgabe letztendlich ausführt, kann sich ändern, kann in ganz andere Speicherbereiche gelegt werden. An der Stelle $FFD2 wird aber immer ein JMP mit der Einsprungadresse stehen. Leider ist diese Sprungtabelle viel zu knapp gehalten. Es gibt so viele interessante ROM-Routinen, die wir alle ohne diese schöne Sicherheit anspringen müssen.

Adresse HEX	Adresse dezimal	Name	Funktion
FF81	65409	CINT	Prüfen der TV-Norm, Berechnung der Taktfrequenz
FF84	65412	IOINIT	Ein/Ausgabe-Reset
FF87	65415	RAMTAS	Prüfen auf freien Basic-RAM
FF8A	65418	RESTOR	Initialisieren der I/O-Vektoren
FF8D	65412	VECTOR	Lesen und Setzen der I/O-Vektoren
FF90	65424	SETMSG	Setzen des Ausgabe-Modus
FF93	65427	SECOND	Ausgeben der Sekundäradresse nach LISTEN
FF96	65430	TKSA	Ausgabe der Sekundäradresse nach TALK
FF99	65433	MEMTOP	Lesen/Setzen des Speicherendes
FF9C	65436	MEMBOT	Lesen/Setzen des Speicheranfangs
FF9F	65439	SCNKEY	Abfragen der Tastatur
FFA2	65442	SETTMO	Setzen der Time-Out-Flagge
FFA5	65445	ACPTR	Zeichen vom seriellen Port in Akku lesen
FFA8	65448	CIOUT	Zeichen vom Akku auf seriellen Port ausgeben
FFAB	65451	UNTLK	Sendet UNTALK an seriellen Bus
FFAE	65454	UNLSN	Sendet UNLISTEN an seriellen Bus
FFB1	65457	LISTEN	Sendet LISTEN an Geräte per seriellen Bus
FFB4	65460	TALK	Sendet TALK an Geräte per seriellen Bus
FFB7	65463	READST	Liest I/O-Status in den Akku
FFBA	65466	SETLFS	Festlegung der Parameter für OPEN
FFBD	65469	SETNAM	Festlegung des Filenamens
FFC0	65472	OPEN	Öffnet spezifizierten File
FFC3	65475	CLOSE	Schließt spezifizierten File
FFC6	65478	CHKIN	Öffnet einen Eingabekanal
FFC9	65481	CHKOUT	Öffnet einen Ausgabekanal
FFCC	65484	CLRCHN	Schließt Ein- und Ausgabekanäle
FFCF	65487	CHRIN	Holt vom aktiven Eingabekanal ein Zeichen in den Akku
FFD2	65490	CHROUT	Sendet Akku-Inhalt auf aktiven Ausgabekanal
FFD5	65493	LOAD	LOAD und VERIFY von Programmen
FFD8	65496	SAVE	Speichern von Programmen
FFDB	65499	SETTIM	Uhrzeit setzen
FFDE	65502	RDTIM	Uhrzeit lesen
FFE1	65505	STOP	STOP-Taste abfragen
FFE4	65508	GETIN	Zeichen aus dem Tastaturpuffer in den Akku lesen
FFE7	65511	CLALL	Schließen aller Kanäle und Files
FFEA	65514	UDTIM	Uhr um 1/60 Sekunde weiterzählen
FFED	65517	SCREEN	Lesen des Bildschirmformates
FFF0	65520	PLOT	Lesen/Setzen der Cursor-Position
FFF3	65523	IOBASE	Lesen der Startadresse der Ein- und Ausgabebausteine

Tabelle 7. Kernel-Routinen

32. Die Urzelle eines Programmprojektes

Wir sind jetzt soweit, daß wir die Urzelle eines Programmprojektes, welches uns eine lange Zeit begleiten wird, aufbauen können. Wir wollen etwas unter den Teppich kehren. Der Teppich, das sind die uns bislang nicht zugängigen RAM-Bereiche unter den ROMs. Haben Sie das nicht auch schon mal erlebt, daß Sie während einer Programmarbeit plötzlich feststellen, Sie benötigen zum Beispiel für eine Zwischenrechnung ein weiteres Programm, oder Sie wälzen Listen und denken sich, ein kleiner Hilfsbildschirm wäre jetzt von Nutzen, oder....

Mit diesem heute zu startenden Programm wäre all das und noch viel mehr realisierbar. Es soll auf einfache Weise beliebige Speicherbereiche unters ROM schieben und sie wieder hervorholen können.

Natürlich braucht die Entwicklung dieses Projektes einige Zeit, zumal wir noch vieles lernen müssen. Deswegen sind wir

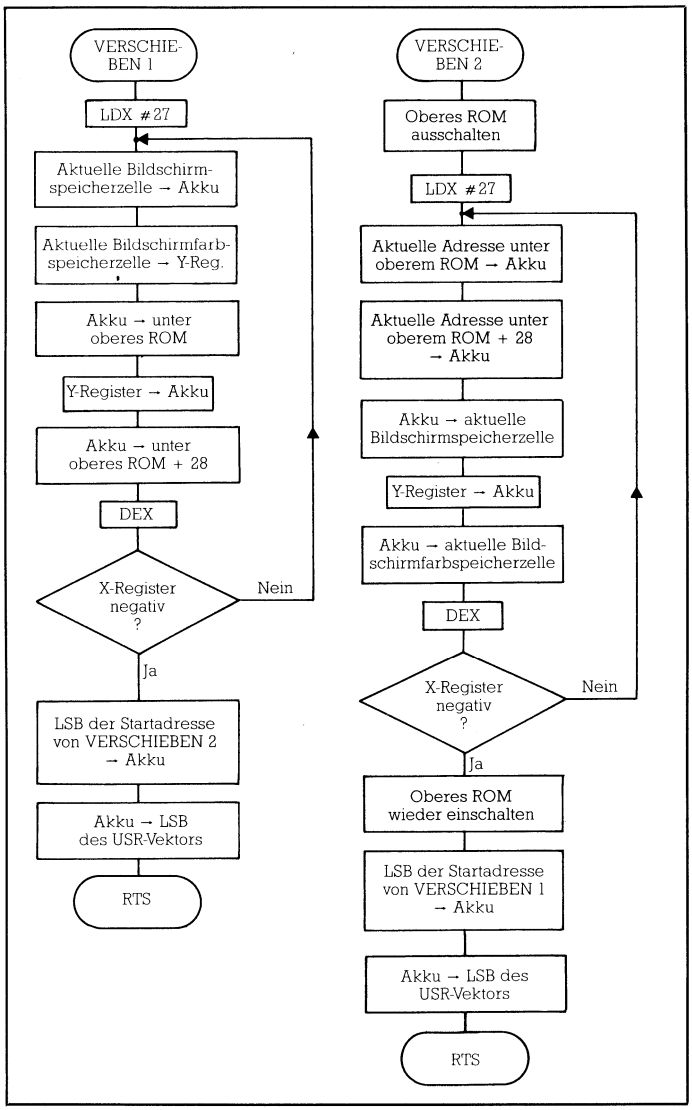


Bild 17. Das Flußdiagramm zu dem im Text erklärten Programm.

in dieser ersten Urzelle noch sehr eingeschränkt: Wir verschieben zuerst einmal nur eine Bildschirm-Kopfzeile unter den oberen ROM-Bereich. Auch in dieser einfachsten Version gibt es noch einige Programmteile, die Sie erst nach der nächsten Ausgabe verstehen werden. Aber irgendwann müssen wir ja mal anfangen, Nägel mit Köpfen zu machen.

Unser Maschinenprogramm soll durch die USR-Funktion aufgerufen werden. Wie wir es in dieser Ausgabe gelernt haben, muß deshalb vor dem ersten Aufruf eine Initialisierung durch Belegen des USR-Vektors mit unserer Startadresse stattfinden. Die Startadresse soll $02B6 (dez. 694) sein, denn dort gibt es einen freien RAM-Bereich bis inklusive $02FF (dez. 767), der weder andere Programme noch Kassettenoperationen stört. Das MSB $02 ist dezimal auch 2 und wird nach 786 gePOKEt:

```
POKE 786,2
```

Das LSB $B6 ist dezimal 182 und soll in 785 geschrieben werden:

```
POKE 785,182
```

Damit ist der USR-Vektor gestellt und wir brauchen uns nicht mehr weiter darum zu kümmern: Jeder USR-Aufruf wird nun den Start des Programmes bewirken. Nun zum Programm selbst. In Bild 17 finden Sie ein Flußdiagramm dazu.

Zunächst konstruieren wir den Teil, der die erste Bildschirmzeile nach $E000 und folgende Speicherstellen schiebt. Das X-Register verwenden wir als Index und laden es mit dez.40 = $27.

Schalten Sie also den SMON ein und starten Sie den Assembler mit:

```
A 02B6
```

Dann geben Sie ein:

```
02B6  LDX   #27
```

Nun packen wir das letzte Zeichen der obersten Bildschirmzeile in den Akku:

```
02B8  LDA   0400,X
```

In das Y-Register legen wir die dazugehörige Farbe aus dem Bildschirmfarbspeicher:

```
02BB  LDY   D800,X
```

Den Akkuinhalt – also die Bildschirminformation – legen wir nach $E000+$27:

```
02BE  STA   E000,X
```

Dasselbe tun wir mit dem Farbcode, der ab $E028+$27 abwärts gespeichert wird. Leider kann man STY nicht X-indiziert-absolut adressieren (siehe Tabelle 5). Deshalb schieben wir zuerst den Y-Registerinhalt in den Akku:

```
02C1  TYA
02C-
2     STA   E028,X
```

Damit ist das letzte Zeichen der Kopfzeile verschoben. Wir zählen das X-Register um 1 herunter:

```
02C-
5     DEX
```

Der X-Index weist nun auf das vorletzte Zeichen, mit dem sich alles ab $02B8 wiederholt. Wenn das X-Register bis 0 heruntergezählt ist, weist es auf das erste Zeichen der Kopfzeile. Die Schleife muß dann noch einmal durchlaufen werden und ein weiteres Herabzählen des X-Registers erzeugt $FF, was zum Setzen der N-Flagge führt. Das ist dann unser Signal, daß die gesamte Kopfzeile übertragen wurde. Die N-Flagge wird durch den BPL-Befehl getestet:

```
02C-
6     BPL   02B8
```

So weit, so gut. Wir hätten natürlich auch das X-Register von 0 an hochzählen können. Zum Beenden der Schleife wäre dann aber ein CPX-Befehl erforderlich gewesen, der jedesmal den X-Registerinhalt mit der Zahl $27 vergleicht.

MERKE: Indexregister in Schleifen abwärts zu zählen, kann Rechenzeit einsparen!

Ab $02CE soll der umgekehrte Vorgang, also das Zurückschieben der vorher gespeicherten Kopfzeile in den Bildschirmspeicher geschehen. Das einfachste wäre es sicherlich, diesen Programmteil mit einem weiteren USR-Kommando zu starten. Das sähe dann so aus:

1.USR-Befehl – schiebt Kopfzeile unter oberes ROM
2.USR-Befehl – holt Kopfzeile zurück in Bildschirmspeicher
3.USR-Befehl – schiebt wieder Kopfzeile unter ROM
4.USR-Befehl – holt sie wieder zurück und so weiter.

Weil aber das Umstellen des USR-Vektors durch POKEs vom Basic aus lästig ist, tun wir das einfach immer am Ende des betreffenden Maschinenprogrammabschnittes. Wir

```
1 REM ********************************    <250>
2 REM *                              *    <229>
3 REM *  TEST FUER DIE 1. VERSION DES *    <139>
4 REM *       PROGRAMM-PROJEKTES     *    <048>
5 REM * V E R S C H I E B E N  V O N *    <009>
6 REM *       SPEICHERBEREICHEN      *    <193>
7 REM *                              *    <234>
8 REM * HEIMO PONNATH    HAMBURG 1984 *    <081>
9 REM ********************************    <002>
10 REM                                     <153>
15 REM ++++++ USR-VEKTOR EINSTELLEN ++++   <065>
20 REM                                     <163>
25 POKE 785,182:POKE 786,2                 <239>
30 REM                                     <173>
35 REM ++++++ KOPFZEILE +++++++++++++++++  <013>
40 REM                                     <183>
45 PRINT CHR$(147)CHR$(18)"TEST
   : BILD $0400=1024, FARBE $D800=55296"CHR$(14
   6)                                      <071>
50 PRINT:PRINT:PRINT"DURCH IRGENDEIN USR-KOMMAN
   DO WIRD NUN IM PROGRAMM-MODUS"          <020>
55 PRINT"DER ERSTE TEIL DES VERSCHIEBE-PROGRAMM
   ES AUFGERUFEN"                          <110>
60 PRINT"DIE KOPFZEILE WIRD UNTER DAS OBERE
    ROM{2SPACE}KOPIERT."                   <215>
65 REM                                     <208>
70 REM ++++++ 1. USR-AUFRUF ++++++++++++   <042>
75 REM                                     <218>
80 A=USR(1)                                <124>
85 PRINT:PRINT"HIER GESCHIEHT DAS DURCH A=USR(1
   ) IN{4SPACE}ZEILE 65"                   <132>
90 PRINT"DABEI IST 1 EIN DUMMY UND MIT A FANGEN
   {2SPACE}WIR AUCH NICHTS WEITER AN."     <063>
95 PRINT"AUF TASTENDRUCK WIRD DER BILDSCHIRM
   {2SPACE}GE-LOESCHT"                     <029>
100 REM                                    <243>
105 REM ++UEBERSCHREIBEN DER KOPFZEILE ++  <046>
110 REM                                    <253>
115 POKE 198,0:WAIT 198,1:PRINT CHR$(147)  <090>
120 REM                                    <007>
125 REM +++ NEUBEGINN DES PROGRAMMES ++++  <173>
130 REM                                    <017>
135 PRINT CHR$(19)"WAS AUCH IMMER JETZT IN DER
    KOPFZEILE{3SPACE}STEHT, ES WIRD BEIM 2.USR"
                        <017>
140 PRINT"VON DEM ZUVOR DURCH DAS ERSTE USR
    GE-{3SPACE}SPEICHERTE UEBERSCHRIEBEN"  <078>
145 PRINT:PRINT"WENN SIE JETZT EINE TASTE DRUEC
   KEN..."                                 <104>
150 POKE 198,0:WAIT 198,1                  <246>
155 REM                                    <042>
160 REM ++++++ 2. USR-AUFRUF ++++++++++    <090>
165 REM                                    <052>
170 A=USR(1):PRINT                         <169>
175 PRINT"IST DIE ALTE KOPFZEILE ZURUECK IN
    DEN{3SPACE}BILDSCHIRMSPEICHER GESCHOBEN."
                        <164>
180 END                                    <052>
```

Bild 18. Test und Demonstration der Verschieberoutine.
Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm

schreiben also das LSB der Programmfortführung ($CE) nach $311. Das MSB bleibt unverändert $02.

```
02C8  LDA   #CE
02CA  STA   0311
02CD  RTS
```

Mit dem RTS sind wir wieder im Basic-Programm gelandet, welches nun normal weiterverarbeitet wird. Erst ein neues USR-Kommando – im Programm oder im Direktmodus – startet den zweiten Teil unseres Maschinenprogrammes (weil in $0311, – der Einsprungpunkt des USR-Befehls – die Startadresse der auszuführenden Routine steht).

In diesem 2. Teil müssen wir erst einige Befehle geben, die Sie jetzt vielleicht noch nicht verstehen. Das hängt damit zusammen, daß zum Herauslesen des RAM unter dem ROM das ROM ausgeschaltet werden muß (entspricht POKE 1,53):

```
02CE  LDA  01
02D0  PHA
02D1  LDA  #35
02D3  STA  01
```

(Der PHA-Befehl dient hier zur Zwischenspeicherung des Akku-Inhaltes). Das ist hiermit geschehen und wir kommen wieder in bekannte Gefilde mit der Ausleseschleife:

```
02D5  LDX  #27
02D7  LDA  E000,X
02DA  LDY  E028,X
02DD  STA  0400,X
02E0  TYA
02E1  STA  D800,X
02E4  DEX
02E5  BPL  02D7
```

Damit ist die gesamte gespeicherte Kopfzeile wieder zurückgeholt und wir können das ROM wieder einschalten:

```
02E7  PLA
02E8  STA  01
```

Befehls-wort	Adressierung	Byte-zahl	Code Hex	Code Dez	Takt-zyklen	Beeinflussung von Flaggen
LDA	absolut,X	3	BD	189	4	N,Z
	0-page-abs,X	2	B5	181	4	N,Z
	absolut,Y	3	B9	185	4	N,Z
LDX	absolut,Y	3	BE	190	4	N,Z
	0-page-abs,Y	2	B6	182	4	N,Z
LDY	absolut,X	3	BC	188	4	N,Z
	0-page-abs,X	2	B4	180	4	N,Z
STA	absolut,X	3	9D	157	5	/
	absolut,Y	3	99	153	5	/
	0-page-abs,X	2	95	149	4	/
STX	0-page-abs,Y	2	96	150	4	/
STY	0-page-abs,X	2	94	148	4	/
INC	absolut,X	3	FE	254	7	N,Z
	0-page-abs,X	2	F6	246	6	N,Z
DEC	absolut,X	3	DE	222	7	N,Z
	0-page-abs,X	2	D6	214	6	N,Z
ADC	absolut,X	3	7D	125	4	N,V,Z,C
	absolut,Y	3	79	121	4	N,V,Z,C
	0-page-abs,X	2	75	117	4	N,V,Z,C
SBC	absolut,X	3	FD	253	4	N,V,Z,C
	absolut,Y	3	F9	249	4	N,V,Z,C
	0-page-abs,X	2	F5	245	4	N,V,Z,C
CMP	absolut,X	3	DD	221	4	N,Z,C
	absolut,Y	3	D9	217	4	N,Z,C
	0-page-abs,X	2	D5	213	4	N,Z,C
BIT	absolut	3	2C	44	4	N,V,Z
	0-page-abs.	2	24	36	3	N,V,Z
CLV	implizit	1	B8	184	2	V
NOP	implizit	1	EA	234	2	/
TAX	implizit	1	AA	170	2	N,Z
TAY	implizit	1	A8	168	2	N,Z
TXA	implizit	1	8A	138	2	N,Z
TYA	implizit	1	98	152	2	N,Z
JMP	absolut	3	4C	76	3	/
	indirekt	3	6C	108	5	/
JSR	absolut	3	20	32	6	/

Tabelle 8. Zusammenfassung aller wichtigen Daten der neuen Befehle

Falls nun wieder ein USR-Kommando auftaucht, soll die Kopfzeile mit dem 1. Programmteil unter das obere ROM gelegt werden wie am Anfang. Wir müssen deshalb den USR-Vektor auf $02B6 zurückschreiben:

```
02EA  LDA   #B6
02EC  STA   0311
02EF  RTS
```

Das wärs! Wenn nun im Programm oder im Direktmodus wieder ein USR-Befehl auftritt, kann das Ganze von vorne beginnen. In dieser Version wird jedesmal eine neue Kopfzeile hin- und wieder zurückgeschoben. Wenn Sie eine einmal festgelegte Kopfzeile immer wieder benutzen möchten, dann stellen Sie den USR-Vektor einfach nicht mehr zurück: Lassen Sie also die Befehle bei 02EA und 02EC weg. Das Programm endet in dem Fall mit:

```
02EA  RTS
```

Eine wichtige Bemerkung noch: So bequem der Ort auch ist, an dem unser kurzes Programm steht, er hat einen gravierenden Nachteil: Falls Sie mittels einer RESET-Taste oder per Software einen Basic-Kaltstart durchführen, geht unser Programm flöten! Dieser Speicherbereich wird im Reset-Programm nämlich mit lauter Nullen überschrieben. Deswegen speichern Sie es bitte bald ab.

In Bild 18 finden Sie ein kleines Testprogramm für unsere Verschieberoutine, und in Tabelle 8 eine Zusammenfassung aller wichtigen Daten der neuen Befehle.

33. Wir stapeln

In Kapitel 28 haben wir beim JSR-Befehl schon den Stapel etwas kennengelernt. Aber so ganz genau wissen wir's ja noch nicht, was das ist. Deswegen jetzt mal im Detail: Der Stapel, auch Prozessorstack genannt, ist der Speicherbereich von dezimal 256 ($100) bis dezimal 511 ($1FF), der direkt von unserer CPU verwaltet wird. Das ist also die gesamte Page 1. Ähnlich wie bei der String-Verwaltung geschieht auch hier das Füllen von oben nach unten. Das erste Byte, welches in den Stack geschoben wird, kommt also nach $1FF, das nächste nach $1FE und so weiter. Voll ist der Stapel, wenn auch $100 besetzt wurde (siehe Bild 19).

Warum heißt das Ding nun eigentlich Stapel? Das erklärt sich aus dem Zugriffs-Prinzip. Man spricht von einer LIFO-Struktur, von »Last In – First Out«, zu deutsch »zuletzt hinein – zuerst heraus«. Das zuerst hineingebrachte Byte befindet sich am Speicherboden ($1FF), das zuletzt eingebrachte an der Speicherspitze. Stellen Sie sich einen Stapel Akten vor (Bild 20).

Offensichtlich wurde der 4. Aktenordner zuletzt auf den Stapel gesteckt. Er kann zuerst heruntergeholt werden. An die Akte 1 kommen wir erst heran, wenn alle anderen heruntergenommen worden sind. Genauso verhält es sich mit dem Prozessorstack: Um an das unterste Byte des Stapels heranzukommen, müssen erst Byte für Byte die darüberliegenden (nach Bild 19 eigentlich die darunterliegenden) weggeschafft werden.

Mit dem Prinzip des Stapelspeichers werden Sie sich auskennen, wenn Sie schon mal andere Programmiersprachen als Basic ausprobiert haben: In Forth beispielsweise operieren Sie ständig mit Stapeln.

Damit wir – und der Prozessor – den Überblick über den Stack behalten, gibt es dankenswerterweise noch einen Stapelzeiger (stackpointer), der jeweils auf den nächsten freien Platz des Stapels weist. Da gibt's nun aber ein kleines Problem: Der Stapel belegt die komplette Seite 1.

Ein Stapelzeiger, der auf zum Beispiel $01FE zeigen soll, müßte das MSB (also 01) und das LSB (also FE) in zwei Bytes lagern. Der Stapelzeiger ist aber nur 8 Bit groß ... Freundlicherweise sorgt unser Mikroprozessor automatisch für das

neunte Bit. Der Zeiger zählt also immer von $FF an rückwärts bis $00 und weist dabei von $1FF bis $100.

Der Stack hat in unserem Computer drei Aufgaben zu erfüllen:
1) Organisation von Unterprogramm-Adressen
2) Zwischenspeicherung bei Unterbrechungen (Interrupts)
3) vorübergehende Datenspeicherung

Die Rolle des Stapels bei Unterprogramm-Aufrufen haben wir in der letzten Folge schon ausgiebig behandelt. Die sogenannten Interrupts heben wir uns noch für später auf – dazu fehlen uns noch ein paar Kenntnisse. Mit der vorübergehenden Speicherung von Daten befassen wir uns gleich, wenn wir an die Befehle zur Stackbehandlung herangehen.

Zuvor – weil das hier gerade ganz gut paßt – noch ein paar Gedanken zur rekursiven Programmierung. Gemeint ist damit eine Programmstruktur, in der sich ein Unterprogramm selbst aufruft. Auch GOSUB-Befehle in Basic bewirken Einträge der Rücksprungadressen im Stapel. Auf diese Weise ergibt sich für unseren Computer eine begrenzte Verschachtelungstiefe bei Unterprogrammaufrufen. Diese wird bei Rekursion besonders schnell erreicht, und das bewirkt die Ausgabe einer OUT OF MEMORY-Fehlermeldung.

34. Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS

Mit dem Stapel haben wir 256 Speicherplätze für eine schnelle Zwischenspeicherung aller möglichen Daten zur Verfügung. Weil der 6510 (und natürlich auch der 6502) diesen Speicherbereich wie die Zeropage behandelt, geht das Speichern sehr schnell. Man muß nur immer die spezielle LIFO-Struktur berücksichtigen.

Im Grunde braucht man eigentlich nur zwei Befehle: Etwas auf den Stapel schieben (in der Literatur oft als Push-Befehl bezeichnet) und etwas herunterziehen, das nennt man dann Pull- oder auch Pop-Befehl.

Unser Prozessor kennt insgesamt sechs auf den Stapel wirkende Anweisungen:

PHA Damit schreibt man den Akku-Inhalt in den Stapel (»PusH-Accumulator«). Der Stapelzeiger wird automatisch eine Position heruntergezählt (er rechnet ja von $FF an abwärts!). Der Inhalt des Akku wird dabei nicht verändert. Deswegen bleibt auch das Status-Register (also die ganzen Flaggen: N V B D I Z C) unbeeinflußt.

PLA »PuLl Accumulator«. Das ist der umgekehrte Weg: Das, was zuoberst auf dem Stapel liegt, wird in den Akku geschrieben. Dadurch wird ein Stapelplatz frei, was den Stapelzeiger veranlaßt, um 1 zu wachsen. Weil das, was da in den Akku geladen wird, 0 sein kann oder auch negativ (also mit gesetztem Bit 7), wird unter Umständen auch die N- oder die Z-Flagge verändert.

Weniger mit Datenzwischenspeicherung haben die anderen Befehle zur Stapel-Manipulation zu tun:

PHP Das steht für »PusH Processor status«, also »schiebe das Prozessor-Status-Register auf den Stapel«. Der aktuelle Flaggenstand kann damit aufbewahrt werden. Das Status-Byte ändert seinen Inhalt dabei ebensowenig wir der Akku bei PHA. Auch hier wird der Stapelzeiger freundlicherweise um 1 herabgezählt.

PLP »PuLl Processor status«, »hole den Prozessor-Status vom Stapel« ist der umgekehrte Befehl, der (wie bei PLA in den Akku) das, was zuoberst im Stapel liegt, in das Flaggen-Register schreibt. Da sollte man höllisch aufpassen, was man damit einlädt: Das ist eine feine Gelegenheit für den Computer, abzustürzen. Der Stapelzeiger wird – wie gehabt – um 1 erhöht.

Nicht direkt mit dem Stapel, sondern mit dem Stapelzeiger befassen sich die beiden folgenden Befehle:

TSX »Transfer Stack-pointer into X«, zu deutsch, »schiebe den Stapelzeiger ins X-Register« eröffnet die Möglichkeit, den Stapelzeiger zu lesen. Dabei bleibt er selbst unverändert erhalten. Weil nun im X-Register alle Werte zwischen $FF und 0 auftreten können, werden auch die Flaggen beeinflußt (N- und Z-Flagge).

TXS Den umgekehrten Weg geht »Transfer X into Stackpointer« = »übertrage X-Register-Inhalt in den Stapelzeiger«. Das ist der einzige Befehl, der es erlaubt, den Stapelzeiger mit einem von uns kontrollierten Wert zu laden. Der Inhalt des X-Registers bleibt dabei unverändert, demzufolge interessieren sich auch die Flaggen nicht dafür.

Alle sechs Anweisungen bestehen nur aus einem Byte und sind implizit adressiert. Die Stapelzeiger-Befehle TXS und TSX benötigen zwei Taktzyklen, die Push-Befehle je drei und die Pull-Befehle vier Taktzyklen zur Bearbeitung.

Es ist etwas schwierig, Stapel-Operationen direkt zu verfolgen. Die meisten Assembler – so anscheinend auch der SMON – gebrauchen ebenfalls diesen Speicherbereich. Verlangt man beispielsweise mit dem SMON-Kommando M 0100 01FF eine Darstellung des Stapelinhaltes, dann findet man eine ganze Menge Spuren der Arbeit des Assemblers. Versucht man die zu löschen oder zu überschreiben, zum Beispiel mit dem nachfolgenden kleinen Programm, dann hat der Assembler die Mühe schon wieder zunichte gemacht, wie man durch erneutes M 0100 01FF schnell sehen kann. Dieses kleine Programm soll unterhalb des durch den Stapelzeiger bezeichneten Bereichs 32 Nullen in den Stapel schreiben:

```
8000  LDA  #00
8002  TSX
```

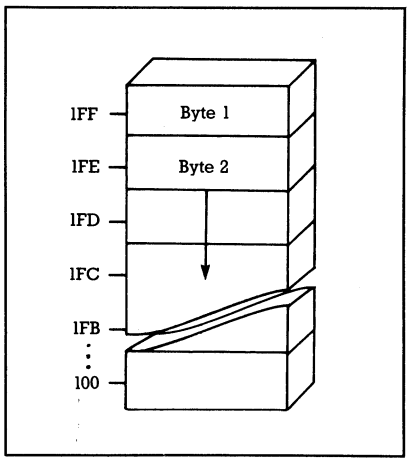


Bild 19. So wird der Stapel gefüllt

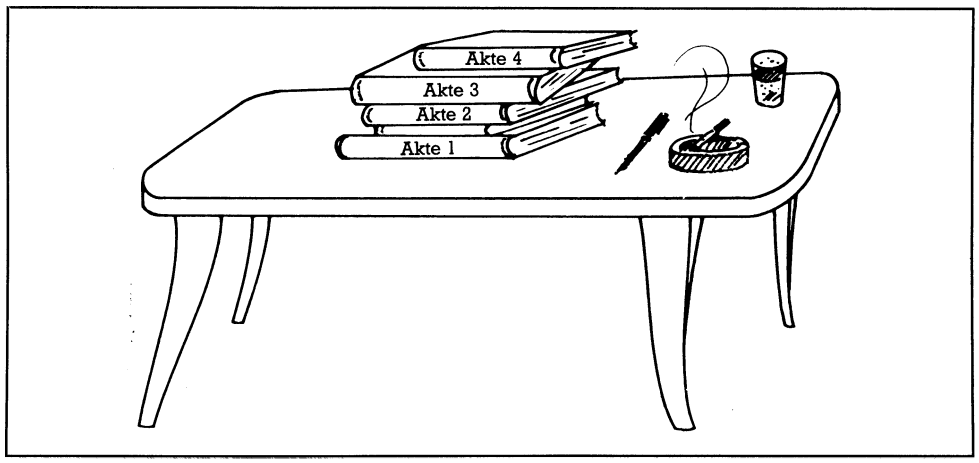


Bild 20. Der Aktenstapel

Der Stapelzeiger wird ins X-Register gerettet.

```
8003  LDY   #20
8005  PHA
```

Wir schieben eine Null auf den Stapel.

```
8006  DEY
8007  BNE   8005
8009  TXS
```

Nach 32 Eintragungen von Nullen stellen wir den alten Stapelzeiger wieder her.

```
800A  BRK
```

Erneutes Kommando M 0100 01FF zeigt keine Nullen. Erst wenn wir anstelle des TXS in Zeile 8009 ein BRK schreiben, den Stapelzeiger also nicht zurückschreiben, erscheinen unsere Nullen. Sieht man genau hin, dann stellt man fest, daß unterhalb des durch den Stapelzeiger bezeichneten Bereichs genau der gleiche Inhalt zu finden ist wie vorher, nur eben mit dem Stapelzeiger verschoben.

Ganz konnte ich dies Rätsel noch nicht lösen, muß ich gestehen, aber für den Gebrauch des Stapels ändert sich dadurch für uns nichts. Worauf muß man achten bei Stapeloperationen? Ganz einfach: Zwischen dem Ablagern eines Wertes auf dem Stapel und dem Zurückholen muß für jeden Push-Befehl ein Pull-Befehl vorhanden sein, für jedes weitere PHA ein PLA, für jedes JSR ein RTS. Nur wenn wir auf diese Symmetrie der Push- und der Pull-Befehle achten (und wie Sie noch aus der vorhergegangenen Ausgabe wissen, sind ja JSR und RTS ebenfalls dazuzurechnen), können wir sicher sein, daß der Stapelzeiger zum Zeitpunkt des Rückholens eines Wertes vom Stapel auch wirklich darauf deutet. Wenn man also nicht ganz genau weiß, wie der verwendete Assembler den Stapel nutzt, sollte man auf Operationen mit den Befehlen TSX und TXS verzichten.

Nun können Sie schon einen Teil der bislang unbekannten Programmsequenz aus der letzten Folge verstehen. Im zweiten Programmteil hatten wir mit

```
02CE  LDA   01
02D0  PHA
```

den Inhalt der Speicherstelle 01 in den Akku geladen und auf den Stapel geschoben. Später – nach einigen weiteren Operationen – wurde dann dieser Speicherinhalt wiederhergestellt durch

```
02E7  PLA
02E8  STA   01
```

Was aber hat es mit dieser Speicherstelle 01 auf sich? Das soll nun als nächstes erklärt werden.

35. Sein oder Nichtsein: Das Rätsel des Prozessorports

Der Commodore 64 hat 64 KByte an RAM zu bieten. Außerdem aber verfügen wir beim normalen Programmieren über weitere 24 KByte, in denen das Betriebssystem, der Basic-Interpreter, Ein- und Ausgabebausteine und der Zeichenspeicher stecken. Wie Sie wissen, umfaßt der Adreßbus aber nur 16 Bit, was uns lediglich 65536 Speicherzellen, also 64 KByte adressieren läßt. Des Rätsels Lösung liegt darin, daß einige Adressenbereiche mehrfach belegt sind. Man kann das vergleichen mit dem Trick des Kastens mit dem doppelten Boden. Welcher Kasteninhalt gerade dem Prozessorzugriff offensteht, wird durch den Prozessorport, das sind die Speicherstellen 00 und 01, gesteuert.

Dr. Helmuth Hauck hat in seiner Serie »Memory Map mit Wandervorschlägen« (64'er, Ausgabe 11 (1984), Seite 135 ff.) die genaue Funktion jedes Bits dieser beiden Speicherstellen erklärt. Wer noch mehr wissen möchte – auch über die Wirkungsweise der beiden Leitungen »Game« und »Exrom« – sollte das nachlesen im »Commodore 64 Programmers Reference Guide« ab Seite 260. Für uns als angehende Assembler-Alchimisten ist die Speicherstelle 1 aber so wichtig, daß wir ganz kurz hier nochmal darauf eingehen.

Die Speichersteuerfunktionen haben die Bits 0 bis 2 der Speicherstelle 1. Je nach Belegung dieser Bits gestaltet sich die 64-KByte-Landschaft unseres Computers wie in Tabelle 9 gezeigt.

Was können wir als Maschinen-Programmierer mit dieser Kenntnis anfangen? Theoretisch stehen uns für unsere Programme damit 64 KByte offen. Praktisch werden wir nur in den seltensten Fällen auf die Ein- und Ausgabe-Bausteine verzichten können. Lassen wir ein reines Maschinenprogramm laufen, ohne jeglichen Rückgriff auf Interpreter oder Betriebssystem, dann haben wir immerhin noch zirka 60 KByte zur freien Verfügung. Benutzen wir Routinen aus diesen beiden ROM-Bausteinen, dann müssen wir sie allerdings – zumindest für den Zeitpunkt des Routineaufrufs – wieder einschalten. Wenn wir – was wohl meistens der Fall sein wird – Kombinationen von Basic- und Assemblersprache verwenden, können wir den gesamten Basic-Speicher bis $A000 frei halten, können auch den bei allen Beispielprogrammen so beliebten Bereich $C000 bis $D000 leer lassen und packen unsere Routinen weitgehend unter die ROMs, die dann jeweils beim Aufruf abgeschaltet werden. So haben wir eine Menge zusätzlichen Speicherplatz ergattert.

Nun können wir auch den letzten Rest des bislang unklaren Programms aus Kapitel 32 verstehen. Nachdem wir den Inhalt der Speicherstelle 1 auf den Stapel gerettet haben (Zeilen $02CE und $02D0), schreiben wir $35 in den Prozessorport:

```
02D1  LDA   #35
02D3  STA   01
```

$35 ist binär 0011 0101. Die Bits 0 bis 2, auf die es uns in diesem Zusammenhang ankommt, bewirken nun das Ausschalten des Interpreters und des Betriebssystems. Die Ein- und Ausgabe-Bausteine bleiben aktiv. Im weiteren Programmverlauf lesen wir die Speicherinhalte ab $E000, wobei wir nun den RAM-Inhalt erfassen. Das sollte vielleicht nochmal klargestellt werden: Jedes Hineinschreiben in die mehrfach belegten Speicherbereiche (dabei sind die Ein- und Ausgabe-Bausteine aber ausgenommen) wird automatisch in den RAM-Bereich umgelenkt. Das ist ja auch klar: In ein ROM kann eben nicht geschrieben werden. Deshalb braucht man dabei die ROMs nicht auszuschalten. Jeder Lesevorgang greift aber auf die ROMs zu, weshalb man sie in unserem Fall ausschalten muß. Wie schon oben beim Stapel erklärt, schalten wir durch das Zurückholen des vorher dorthin geretteten alten Inhalts der Speicherstelle 1 in den Prozessorport wieder den Normalzustand ein.

36. Die indirekte Adressierung

Wir werden nun die beiden letzten noch ausstehenden Arten der Adressierung kennenlernen. Beides sind indirekte Adressierungsarten. Mit dem indirekten JMP-Befehl (zum Beispiel

Speicherstelle 1 Bits 2	1	0	$A000-$BFFF	$D000-$DFFF	$E000-$FFFF
1	1	1	Basic	I/O	Kernel
1	1	0	RAM	I/O	Kernel
1	0	1	RAM	I/O	RAM
1	0	0	RAM	RAM	RAM
0	1	1	Basic	Zeichen	Kernel
0	1	0	RAM	Zeichen	Kernel
0	0	1	RAM	Zeichen	RAM
0	0	0	RAM	RAM	RAM

Tabelle 9 zeigt, welche Bausteine bei verschiedener Belegung der Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind. (Frei nach Dr. Hauck, 64'er Ausgabe 11/84, Seite 136)

JMP(0300)) sind wir in Kapitel 28 schon vertraut geworden. Wir hatten auch gelernt, daß es sich hierbei um einen absoluten Einzelgänger handelt, der nur für so einen Sprung erlaubt ist. Ebenso haben wir die indizierte Adressierung zu beherrschen gelernt: Das war die Sache mit den Indexregistern X oder Y. Eine Kombination aus beiden (also der indirekten und der indizierten) Adressierungsarten sind die indiziert-indirekte und die indirekt-indizierte Adressierung.

Die indirekt-indizierte Adressierung

Fangen wir mit der sehr häufig benutzten indirekt-indizierten Adressierung an: Man nennt sie auch »indirekt Y« oder »nach-indizierte indirekte« Adressierung. Am besten sehen wir uns mal so einen Befehl an:

```
LDA     (FA),Y
```

Die Klammer erinnert uns an den indirekten JMP-Befehl. Tatsächlich hat sie hier auch dieselbe Funktion: In FA und FB steht ein Zeiger auf eine Adresse. Nehmen wir mal an, die Belegung der Speicher wäre:

```
FA      01
FB      80
```

und im Y-Register stünde eine 5. Der Zeiger FA/FB weist also auf die Speicherstelle 8001. Da haben wir also wieder das Prinzip des toten Briefkastens. Der Computer guckt in den hohlen Baum FA/FB (LSB in FA, MSB in FB) und findet dort die Treffpunktadresse. Nun sind diese toten Briefkästen aber auch den gegnerischen Alchimisten-Agenten bekannt. Es kommt also noch ein Trick dazu: Zur dort aufgefundenen Adresse wird der Inhalt des Y-Registers addiert. In unserem Fall fanden wir also in FA/FB die Adresse 8001, im Y-Register steht eine 5, somit ist die endgültige Adresse 8001+5 = 8006. Unser Beispiel »LDA(FA),Y« bewirkt daher, daß in den Akku der Inhalt der Speicherstelle 8006 geladen wird. Nachindiziert nennen manche die Adressierung deswegen, weil zunächst dem Zeiger nachgegangen wird, der in unserem Beispiel auf 8001 weist, und erst danach durch Addition des Inhalts des Y-Registers die endgültige Speicherstelle (hier also 8006) berechnet wird.

Als Zeiger (also die Adresse in der Klammer) sind nur Zeropagespeicherstellen verwendbar, als Indexregister darf man hier nur das Y-Register gebrauchen. Von den bisher behandelten Befehlen können ADC, CMP, LDA, SBC und STA mit dieser Adressierungsart verwendet werden. Genaueres finden Sie wieder in der Tabelle mit der Befehlsübersicht (Tabelle 10).

Bevor wir uns dem anderen indirekten Adreß-Modus zuwenden, wollen wir uns überlegen, wozu man die indirekt-indizierte Adressierung verwendet. Wie Sie sich natürlich erinnern können, konnte man mit der normalen indizierten Adressierung, zum Beispiel mit

```
LDA 8000,Y
```

durch Variation des Indexregisters (hier das Y-Register) 256 Speicherstellen erfassen (Y von FF herunter bis 00). Will man mehr als diese 256 berücksichtigen, dann muß eine neue Basis (im Beispiel also anstelle der 8000) gewählt werden. Um das zu illustrieren, sehen wir uns mal den Anfang eines Programms an, welches den gesamten Bildschirminhalt ausliest und nach E000 schreibt:

```
1000    LDY     #00
1002    LDA     0400,Y
1005    STA     E000,Y
1008    LDA     0500,Y
100B    STA     E100,Y
100E    LDA     0600,Y
1011    STA     E200,Y
1014    LDA     0700,Y
1017    STA     E300,Y
101A    DEY
101B    BNE     1002
...
```

Wie Sie sehen, erfordert das durch die Tatsache, daß vier Blöcke zu je 256 Bytes übertragen werden müssen, immerhin schon 28 Bytes Programmtext. Nun soll die indirekt-indizierte Adressierung verwendet werden, um dieselbe Aufgabe zu lösen. Wir legen zunächst zwei Zeiger auf der Zeropage fest:

FA/FB sollen die Bildschirmadresse enthalten
FC/FD die Zieladresse ab E000.

```
1000    LDA     #00
1002    STA     FA
1004    STA     FC
```

Das waren die LSBs der Zeiger, es folgen die MSBs:

```
1006    LDA     #04
1008    STA     FB
100A    LDA     #E0
100C    STA     FD
```

Damit sind die Zeiger festgelegt. Es sind vier Blöcke zu je 256 Bytes zu übertragen. Diese Blockanzahl legen wir ins X-Register als Zähler:

```
100E    LDX     #04
```

Dann laden wir ins Y-Register ebenfalls einen Zähler (den Index):

```
1010    LDY     #00
```

Jetzt kann die eigentliche Übertragungsschleife starten:

```
1012    LDA     (FA),Y
1014    STA     (FC),Y
1016    DEY
1017    BNE     1012
```

Wenn das Y-Register wieder bei 0 angekommen ist (von der ersten 0 nach einem Unterlauf über FF, FE und so weiter bis 0), ist der erste Block übertragen. Wir erhöhen nun das MSB beider Zeiger um 1:

```
1019    INC     FB
101B    INC     FD
```

Außerdem zählen wir den Blockzähler um 1 herunter:

```
101D    DEX
101E    BNE     1012
...
```

Wenn das Programm auf diese Weise auch drei Byte mehr Speicherplatz braucht, ist doch leicht der Vorteil zu sehen: Müssen wir nämlich (statt nur vier) mehr Blöcke übertragen (bis zu 255), dann verändert sich unser zweites Programm um keinen Deut (außer dem Zähler im X-Register, der nun mit der jeweils anderen Block-Anzahl geladen wird), während die

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
LDA	indirekt X	2	A1	161	6	N,Z
	indirekt Y	2	B1	177	5*	N,Z
STA	indirekt X	2	81	129	6	—
	indirekt Y	2	91	145	6	—
ADC	indirekt X	2	61	97	6	N,V,Z,C
	indirket Y	2	71	113	5*	N,V,Z,C
SBC	indirekt X	2	E1	225	6	N,V,Z,C
	indirekt Y	2	F1	241	5*	N,V,Z,C
CMP	indirekt X	2	C1	193	6	N,Z,C
	indirekt Y	2	D1	209	5*	N,Z,C
PHA	implizit	1	48	72	3	—
PLA	implizit	1	68	104	4	N,Z
PHP	implizit	1	08	8	3	—
PLP	implizit	1	28	40	4	alle
TSX	implizit	1	BA	186	2	N,Z
TXS	implizit	1	9A	154	2	—

* Wenn bei der Befehlsausführung eine Page-Grenze überschritten wird, muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 10. Übersicht der in dieser Folge vorgestellten Befehle

erste Programmtechnik für jeden weiteren Block um sechs Bytes erweitert werden muß.

Es gibt noch eine ganze Reihe von Anwendungsmöglichkeiten, die die indirekt-indizierte Adressierung so attraktiv machen. Für Geschwindigkeitsfanatiker (ich selbst bin bei Grafik-Fragen auch einer!) muß aber gesagt werden, daß dem Speicherplatzvorteil ein Geschwindigkeitsnachteil gegenübersteht. Jeder indirekt-indiziert adressierte Befehl braucht einen Taktzyklus länger als der vergleichbare absolut-indizierte Befehl. Zu diesen Feinheiten werden wir aber später noch kommen.

Die indiziert-indirekte Adressierung

Wenden wir uns nun der letzten noch fehlenden Adressierungsart zu, der indiziert-indirekten. Man nennt sie auch »vorindizierte indirekte« oder »indirekt X« Adressierung. Sehen wir auch hier zunächst ein Beispiel an:

```
STA     (FA,X)
```

Auch hier drückt die Klammer wieder aus, daß der Klammerinhalt ein Zeiger ist. Das ist jetzt aber nicht das Bytepaar FA/FB, sondern zur angegebenen Adresse FA soll noch der Inhalt des X-Registers addiert werden. Nehmen wir mal an, dort stünde eine 2, dann wird der Zeiger FC/FD mit diesem Befehl angesprochen, denn FA+2=FC und entsprechend FB+2=FD. Wenn in den Speicherstellen FA bis FF folgender Inhalt zu finden ist:

```
00FA   00
00FB   04      FA/FB = 0400
00FC   00
00FD   E0      FC/FD = E000
00FE   10
00FF   80      FE/FF = 8010
```

dann könnte das eine ganze Tabelle von Zeigern sein, die jeweils durch den X-Registerinhalt angesprochen werden. Der Akkuinhalt wird in unserem Beispiel nach 0400 geschrieben, wenn im X-Register 0 steht, nach E000, wenn das X-Register eine 1 enthält und nach 8010, wenn stattdessen eine 2 im X-Register zu finden ist.

Sie werden sich vielleicht auch bei diesem Beispiel gefragt haben, was passiert, wenn im X-Register unseres Beispiels eine 6 steht. Nun, unser 8-Bit-Prozessor läuft über, und wir finden einen Zeiger 00/01.

Rein theoretisch ist diese Adressierungsweise ganz interessant. Aber auf der Zeropage ist's reichlich eng, und nur selten kommt man daher in die Lage, dort eine Zeigertabelle einzurichten, die man mittels des X-Registerinhalts und der indiziert-indirekten Adressierung abgreifen kann. Die Bedeutung dieser Adressierungsart ist also nur recht gering. Außerdem erfordert sie sechs Taktzyklen zur Bearbeitung und ist somit auch noch recht langsam. Von den bisher bekannten Befehlen sind die folgenden damit verwendbar: ADC, CMP, LDA und STA.

Bevor wir die Adressierung zu den Akten legen, sei noch erwähnt, daß manche Lehrbücher noch eine weitere Art, die Akkumulator-Adressierung, unterscheiden. Betroffen sind davon vier 1-Byte-Befehle, die wir noch kennenlernen werden und die man ebensogut als implizit adressiert ansehen kann.

37. Die ersten Kernel-Routinen

Sicher werden Sie alle schon von der Kernel-Routine FFD2 gehört haben und sie vielleicht auch schon verwenden. Wenn nicht, um so besser, denn dann sind Sie noch nicht vom einseitigen Gebrauch dieses Instruments verdorben. Die meisten Kernel-Adressen sind nämlich sehr vielseitig verwendbar, je nach den Vorgaben. Das ist wie mit einem Haushaltsgerät, das immer nur zum Rühren von Kuchenteig eingesetzt wird. Dabei kann man damit auch noch Saft machen, Gurken schnitzeln, Getränke mixen ... Genauso wie man in diesem etwas schiefen Vergleich die Gebrauchsanleitung kennen sollte, um die ganzen anderen Funktionen ausnutzen zu können, muß man hier noch einige Dinge über die Kernel-Aufrufe beherzigen.

Für jede Verwendung der Kernel-Sprungtabelle sollte man sich angewöhnen, dies in drei Schritten zu tun:

1) die nötigen Vorbereitungen treffen
2) Routineaufruf
3) Fehlerabfrage und -behandlung

Fangen wir mit dem Punkt »Vorbereitungen« an. Einige Routinen brauchen Informationen, die ihnen erst durch andere Kernel-Routinen beschafft werden. Ruft man diese anderen Routinen vorher nicht auf, dann funktioniert auch der erwünschte Aufruf nicht richtig. Wenn die Routine einen bestimmten Wert im Y-Register erwartet, dann muß der dort auch stehen. Wenn nicht, dann geht das Programm in die Hose. Bei jeder Kernel-Routine, die hier beschrieben wird, gebe ich alle nötigen Vorbereitungen an.

Der Routinenaufruf sollte immer mittels JSR erfolgen. Alle auf diese Weise aus der Kernel-Sprungtabelle abzurufenden Programme enden nämlich mit einem RTS. Damit keine wichtigen Werte aus dem Aufrufprogramm überschrieben werden, man sie also vor dem Aufruf der Kernel-Routine irgendwohin retten kann, gebe ich auch noch an, welche Register durch die Routine verändert werden und wieviel Stapelspeicherplatz bereitgehalten werden muß.

Die Routinen sind so konstruiert, daß beim Auftreten eines Fehlers nach der Rückkehr das Carry-Bit gesetzt ist. Durch Untersuchen des Carry können so Fehler rechtzeitig erkannt und behandelt werden. Im Akku findet man in dem Fall dann eine Fehlernummer. Die Ausgabe der Fehlermeldung erfolgt also nicht – wie im Basic – in Klarschrift. In Tabelle 11 sind die Fehlernummern und ihre Bedeutung aufgelistet.

Welche Fehlernummern eine Routine ausgeben kann, wird ebenfalls von mir bei jeder Routinen-Besprechung angegeben.

Nun aber zur ersten Routine FFD2, die wie einen Rattenschwanz eine Reihe weiterer nach sich zieht:

Name	CHROUT
Zweck	Ausgabe eines Zeichens
Adresse	$FFD2 dez. 65490
Vorbereitungen	(CHKOUT,OPEN) Zeichen im Akku
Fehler	0
Stapelbedarf	8
Register	Akku

Falls Sie diese Routine schon einmal benutzt haben, dann geschah es vermutlich ohne die Vorbereitungen CHKOUT und OPEN. Freundlicherweise hat unser Computer einige Voreinstellungen schon für uns getroffen. Denn normaler-

Nummer	Text	Bedeutung
0	BREAK	Während des Programms wurde die RUN/STOP-Taste gedrückt
1	TOO MANY FILES	Man kann maximal 10 offene Files einrichten
2	FILE OPEN	Ein bereits geöffnetes File wird nochmals geöffnet
3	FILE NOT OPEN	Auf ein noch nicht geöffnetes File sollte zugegriffen werden
4	FILE NOT FOUND	Das geforderte File ist nicht verfügbar
5	DEVICE NOT PRESENT	Das angesprochene Gerät zeigt keine Reaktion
6	NOT INPUT FILE	Aus einem Schreibfile kann nicht gelesen werden
7	NOT OUTPUT FILE	In ein Lesefile kann nicht geschrieben werden
8	MISSING FILE NAME	Bei Operationen, die einen Filenamen erfordern, fehlt dieser
9	ILLEGAL DEVICE NUMBER	Das versuchte Kommando ist beim angesprochenen Gerät nicht möglich

Tabelle 11. Fehlernummern und ihre Bedeutung. Die Nummern findet man bei gesetztem Carry im Akku.

weise sendet CHROUT ein Zeichen über einen schon geöffneten Ausgabekanal, und der ist zum Bildschirm geschaltet. Ein kleines Beispielprogramm soll das illustrieren. Zunächst laden Sie bitte den SMON ein und starten Sie ihn. Nun soll eine Texttabelle angelegt werden. Das funktioniert beim SMON am bequemsten über das K-Kommando. Geben Sie ein K 6000. Der SMON antwortet mit:

'6000 . . . . . .

Wenn Sie nun die RUN/STOP-Taste drücken, können Sie mit dem Cursor in diese Punktzeile fahren und einen Text schreiben:

'6000 HALLO ASSEMBLER-ALCHIMIST . . . . . .

Sinnvoll – vor allem für die weitere Verwendung dieses Textes – ist es, ein (RETURN), also dezimal 13 oder $0D anzuschließen. Dazu gibt es natürlich den Weg über den Assemblerbefehl. Zur Übung wollen wir aber das M-Kommando verwenden. Geben Sie ein (zuerst die »RETURN«-Taste betätigen) M6018, dann wieder RUN/STOP, und fahren Sie mit dem Cursor auf Speicherstelle 601A (falls Sie in 6019 kein Leerzeichen $20 stehen haben, dann fügen Sie's jetzt noch ein). Geben Sie nun anstelle des dort stehenden Bytes 0D ein, und drücken Sie die RETURN-Taste. Der Monitor sollte jetzt zeigen:

```
:6018 54 20 0D
```

etc.

Unser Text soll mit einem BRK enden. Deshalb gehen wir jetzt in den Assembler-Modus mit dem SMON-Kommando A 601B und schreiben:

```
601B   BRK
```

Nun folgt das eigentliche Prográmmchen, das Byte für Byte bis zur Null (BRK) den Text aus der gerade erstellten Texttabelle liest und mittels FFD2 auf den Bildschirm bringt:

```
601C   LDY    #00
601E   LDA    6000,Y
6021   BEQ    602C
```

Das Y-Register wird als Index initialisiert, dann die Texttabelle in den Akku geladen. Wenn das Programm dabei auf die Null stößt, verzweigt es zum Ende. Jetzt folgt die Routine zur Bildschirmausgabe:

```
6023   JSR    FFD2
6026   BCS    602D
```

Falls bei der Kernal-Routine etwas schiefgelaufen ist, wird das Carry-Bit gesetzt, was wir überprüfen und zu einem BRK-Kommando verzweigen (das ist natürlich nur sinnvoll, solange ein Monitor oder Assembler wie der SMON aktiv ist). Nun erhöhen wir das Index-Register und das ganze beginnt von vorne:

```
6028   INY
6029   JMP    601E
602C   RTS
602D   BRK
```

Wenn wir nun aus dem SMON mit F und anschließendem X aussteigen und ein kleines Basic-Aufrufprogramm machen (Bei OUT OF MEMORY ERROR bitte NEW eingeben):

```
10 PRINTCHR$(147)
20 SYS 24604 :REM = $601C
30 END
```

dann können wir uns die Wirkung unseres Programms ansehen: Nach RUN wird der Bildschirm gelöscht und unser Text ausgedruckt.

FFD2 nimmt uns also eine Menge Arbeit ab: Automatisch legt diese Routine in den Bildschirmspeicher den Bildschirmcode (sie rechnet also auch gleich ASCII, das wir ja eingegeben haben, in den POKE-Code um) und in die dazugehörige Bildschirmfarbspeicherstelle den aktuellen Farbcode. Sie setzt außerdem noch den Cursor weiter.

Mit FFD2 kann man aber noch viel mehr machen! Schließlich ist ja der Bildschirm (Gerätenummer 3) nicht der einzige mögliche Empfänger. Wir wollen als nächstes mal eine Ausgabe mittels FFD2 auf den Drucker erzielen. Hier sind die Vorbereitungen allerdings nötig. Zunächst mal müssen wir uns noch zwei weitere Kernal-Routinen ansehen, nämlich CHKOUT und OPEN.

Name	CHKOUT
Zweck	Kanal zum Ausgang definieren
Adresse	$FFC9 dez. 65481
Vorbereitungen	OPEN log. Filenummer ins X-Register
Fehler	0,3,5,7
Stapelbedarf	4
Register	Akku, X-Register

Mit dieser Routine kann jedes File, der zuvor durch OPEN spezifiziert worden ist, zum Ausgabefile erklärt werden. Natürlich muß dann das derart angesprochene Gerät auch ein Ausgabegerät sein. Andernfalls ergibt sich ein Fehler. Bevor man Daten über einen Kanal senden will, muß CHKOUT durchgeführt werden. Wenn die mittels OPEN übergebene Geräteadresse größer als 3 ist, sendet diese Routine automatisch auch ein LISTEN-Kommando an das Ausgabegerät. LISTEN setzt dann zum Beispiel den Drucker in Empfangsbereitschaft. Die Durchführung von CHKOUT ist einfach (vorausgesetzt, man hat vorher OPEN aufgerufen): In das X-Register wird die logische Filenummer geschrieben und dann per JSR FFC9 (CHKOUT) angesteuert.

Nun zur anderen Vorbereitung von FFD2, zu OPEN:

Name	OPEN
Zweck	Öffnen eines logischen Files
Adresse	$FFC0 dez. 65472
Vorbereitungen	SETLFS,SETNAM
Fehler	1,2,4,5,6
Register	Akku, X- und Y-Register

Die Routine OPEN an sich anzusprechen ist relativ einfach. Es genügt ein JSR FFC0. Zuvor allerdings – der Rattenschwanz wird länger – muß mit SETNAM der Filename und mit SETLFS die logische Filenummer, die Geräteadresse und eventuell eine Sekundäradresse festgelegt sein. Erst danach kann das File geöffnet werden durch OPEN. Also sehen wir uns noch SETLFS und SETNAM an:

Name	SETLFS
Zweck	Spezifikationen eines logischen Files
Adresse	$FFBA dez. 65466
Vorbereitungen	logische Filenummer in Akku Gerätenummer ins X-Register Sekundäradresse ins Y-Register
Fehler	keine
Stapelbedarf	2
Register	keine

SETLFS legt für die anderen Kernal-Routinen logische Filenummer, Gerätenummer und Sekundäradresse fest. Die logische Filenummer ist dabei eine Schlüsselzahl, die in eine durch OPEN angelegte File-Tabelle weist. Die Gerätenummer kann zwischen 0 und 31 liegen, dabei sind folgende Zuordnungen vorgesehen:

```
0 Tastatur          2 RS232C-Kanal
1 Datasette         3 Bildschirm
```

Gerätenummern ab 4 beziehen sich automatisch auf Geräte am seriellen Bus. Dabei gilt im allgemeinen:

```
4 Drucker
8 Diskettenstation
```

Die Sekundäradresse ist eine Kommandonummer, die für das jeweils angesprochene Gerät spezifisch ist, zum Beispiel 10 bewirkt beim Drucker Commodore 1526, daß das Gerät in die Grundstellung geht (siehe jeweiliges Handbuch). Will man keine Sekundäradresse verwenden, dann muß FF ins Y-Register geschrieben werden. Der Aufruf von SETLFS geschieht also in folgender Weise: In den Akku lädt man die gewünschte logische Filenummer, ins X-Register die Geräteadresse und ins Y-Register FF oder aber die Sekundäradresse. Danach erfolgt der Sprung mit JSR FFBA.

Schließlich noch zu SETNAM:

Name	SETNAM
Zweck	Filenamen festlegen
Adresse	FFBD dez. 65469
Vorbereitungen	Namenslänge in den Akku LSB des Namenstextes in X-Register MSB des Namenstextes in Y-Register
Fehler	keine
Stapelbedarf	2
Register	Akku, X- und Y-Register

Vor der Eröffnung eines Files mittels OPEN muß diese Routine den Filenamen festlegen. Dazu schreibt man in den Akku die Länge des Namens und in die Register X, Y die Startadresse (LSB ins X-Register, MSB ins Y-Register) der Namenstext-Tabelle. Der Ort dieser Tabelle ist frei wählbar. Wird kein Filename gewünscht, dann gibt man dem Akku die Länge 0 an. X- und Y-Register sind in dem Fall ohne Bedeutung.

Damit – sollte man meinen – hätten wir nun alle Bedingungen erfüllt, FFD2 zur Ausgabe auf den Drucker zu bewegen. Leider ist das noch nicht der Fall: FFD2 schließt nämlich das File und den Ausgabekanal nicht. Das kann – wenn man's nicht beachtet – zu Fehlern oder zur weiteren Ansprache des Druckers führen, auch wenn die gar nicht mehr erwünscht ist. Deswegen sollten noch zwei Kernel-Routinen angehängt werden, von denen die eine (CLRCHN) alle Ein- und Ausgabekanäle wieder in den Ausgangszustand zurückführt, und die andere (CLOSE) das File ordnungsgemäß schließt:

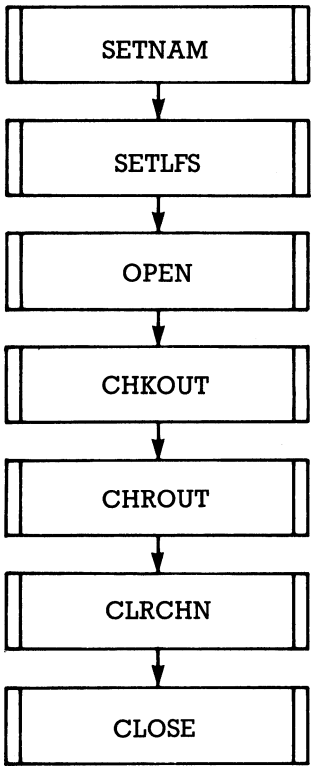
Name	CLRCHN
Zweck	Ein- und Ausgabekanäle in Ausgangsstellung bringen
Adresse	$FFCC dez. 65484
Vorbereitung	keine
Fehler	keine
Stapelbedarf	9
Register	Akku, X-Register

Der Aufruf von CLRCHN erfolgt einfach durch JSR FFCC. Die Wirkung ist enorm: Mit einem Schlag werden alle Kanäle freigeräumt. Eingangskanälen wird ein UNTALK (dem Gerät wird gesagt: Halt den Mund), Ausgangskanälen ein UNLISTEN (das bedeutet soviel wie: Hör nicht mehr zu) übermittelt. Der Ausgangszustand stellt sich wieder her: Tastatur als Eingabe-Bildschirm als Ausgabegerät.

Die endgültig letzte Routine für diesmal ist CLOSE:

Name	CLOSE
Zweck	Schließen logischer Files
Adresse	$FFC3 dez. 65475
Vorbereitungen	logische Filenummer in Akku
Fehler	0
Stapelbedarf	2
Register	Akku, X- und Y-Register

Bild 21. Die Abfolge der Routineaufrufe ▶



Wenn für ein File alle Ein- und Ausgabeoperationen beendet sind, kann es – nach Einschreiben der Filenummer in den Akku – mittels CLOSE ordnungsgemäß geschlossen werden. Der Eintrag in der Filetabelle wird auf diese Weise gelöscht.

So, jetzt sind wir soweit, daß wir die Textausgabe auf dem Drucker programmieren können. Bild 21 faßt die einzelnen Schritte nochmal zusammen.

Und hier das Programm dazu. Wir verwenden die im anderen Beispiel schon aufgebaute Texttabelle weiter. Zunächst also SETNAM:

```
601C   LDA    #00
601E   JSR    FFBD
6021   BCS    6053
```

Wenn ein Fehler aufgetreten ist, findet man ein gesetztes Carry-Bit. In dem Fall wird verzweigt zu einem BRK-Kommando (was die Anwesenheit eines Monitors erforderlich macht, solange man sich noch nicht sicher ist, ob Fehlermeldungen auftauchen). Die Null im Akku besagt, daß kein Filename gewünscht ist. Dann kommt SETLFS:

```
6023   LDA    #04
6025   LDX    #04
6027   LDY    #FF
6029   JSR    FFBA
602C   BCS    6053
```

Es wurde ein File festgelegt mit der logischen Filenummer 4, der Geräteadresse 4 und ohne Sekundäradresse. Jetzt geben wir das OPEN-Kommando:

```
602E   JSR    FFC0
6031   BCS    6053
```

Der Ausgabekanal wird definiert mit CHKOUT:

```
6033   LDX    #04
6035   JSR    FFC9
6038   BCS    6053
```

Damit sind alle Vorbereitungen erledigt und die Zeichenausgabe kann wie im ersten Programm durchgeführt werden mit CHROUT:

```
603A   LDY    #00
603C   LDA    6000,Y
603F   BEQ    604A
6041   JSR    FFD2
6044   BCS    6053
6046   INY
6047   JMP    603C
```

Alle Zeichen sind nun ausgedruckt. Wir rufen CLRCHN auf:

```
604A   JSR    FFCC
```

Als letzte Routine folgt nun noch CLOSE:

```
604D   LDA    #04
604F   JSR    FFC3
6052   RTS
```

Damit wurde das File Nummer 4 geschlossen. Anschließend erfolgte der Rücksprung aus dem Programm. Für die

Fehlerbehandlung habe ich nur einen BRK vorgesehen, der sofortigen Registerüberblick erlaubt, wenn zum Beispiel der SMON im Speicher enthalten ist.

6053 BRK

Ohne Monitor im Speicher kann der Computer allerdings abstürzen oder im besten Fall einen Basic-Warmstart durchführen. Wenn Sie sowas also für Ihre Zwecke programmieren möchten, sollten Sie einen anderen Weg suchen, die Fehler aufzufangen. Man hat ja nicht immer einen Monitor eingeladen.

Mit diesen sieben Kernel-Routinen beenden wir dieses Kapitel. In der 64'er, Ausgabe 12/84 haben B. Schneider und K. Schramm in ihrer Serie »In die Geheimnisse der Floppy eingetaucht« gezeigt, wie man mittels der besprochenen Routinen, und einiger anderer, auch die Diskettenstation ansprechen oder sogar Floppy und Drucker zum »Spooling« veranlassen kann. Das habe ich zwar schon öfter gesagt, muß es aber trotzdem immer wieder tun: Durch das Nachvollziehen fremder Programme kann man sehr viel lernen.

38. Der C 64 und Fließkommazahlen

Inzwischen wissen Sie ja, daß alle Daten im Computer im Binärformat enthalten sind. Wie man eine normale, ganze Zahl zur binären umrechnet, wurde schon im Grafik-Kurs (64'er, Ausgaben 4 und 5 von 1984) gezeigt. Da aber viele Leser dieses Assemblerkurses die genannten Ausgaben nicht besitzen, soll doch nochmal vorgestellt werden, welcher Rechenweg der einfachste ist. Als Beispiel nehmen wir die Zahl 1985. Man teilt diese Zahl so lange durch 2, bis das Ergebnis 0 wird. Jedesmal notiert man sich den Rest, der entweder 0 oder 1 sein kann:

```
1985 : 2=  992 Rest 1
 992 : 2=  496 Rest 0
 496 : 2=  248 Rest 0
 248 : 2=  124 Rest 0
 124 : 2=   62 Rest 0
  62 : 2=   31 Rest 0
  31 : 2=   15 Rest 1
  15 : 2=    7 Rest 1
   7 : 2=    3 Rest 1
   3 : 2=    1 Rest 1
   1 : 2=    0 Rest 1
```

Auch wenn Sie es noch nicht erkennen: Da steht schon das binäre Ergebnis. Von unten nach oben gelesen, ist das nämlich der Rest:

111 1100 0001

Nun reden wir ja von Fließkommazahlen. Also verändern wir unser Beispiel noch etwas. Jetzt soll uns die Zahl 1985,125 interessieren. In der Ausgabe 6/84 haben Sie gelernt, daß man das Komma verschieben kann, um daraus beispielsweise $1{,}985125 \times 10^3$ zu machen. Wir wollen uns das Verschieben des Kommas aber für etwas später aufheben und zunächst einmal außer dem schon umgewandelten Vorkammateil nun auch den Nachkommateil, also die 0,125, ins Binärformat übertragen.

Genauso, wie wir vorhin eine Kettendivision durch 2 verwendet haben, gebrauchen wir nun eine Kettenmultiplikation mit 2. Der gesamte Nachkommateil wird dabei verdoppelt. Entweder ergibt sich dabei eine Vorkommastelle (das ist dann immer eine 1) oder das Ergebnis bleibt kleiner als 1. Wenn sich bei einem solchen Rechenschritt keine Vorkommastelle ergibt, schreibt man an die entsprechende Nachkommastelle der Binärzahl eine 0, andernfalls eine 1. Es wird solange verdoppelt, bis keine Nachkommastellen mehr zur Verfügung stehen. Das klingt ziemlich umständlich. Am besten sehen Sie sich das jetzt mal an unserem Beispiel an:

0,125 x 2 = 0,250 1. Nachkommastelle:0

Beim ersten Verdoppeln hat sich keine neue Vorkommastelle ergeben, deshalb ist die erste Nachkommastelle der Binärzahl eine Null.

0,25 x 2 = 0,5 2. Nachkommastelle:0

Auch beim zweiten Verdoppeln ermitteln wir keine neue Vorkommastelle, wodurch sich wieder eine Null als Nachkommastelle ergibt.

0,5 x 2 = 1,0 3. Nachkommastelle:1

Hier hat sich nun eine Vorkommastelle beim Verdoppeln gebildet: Daher taucht als 3. Nachkommastelle unserer Binärzahl eine 1 auf. Gleichzeitig war das die letzte Nachkommastelle, denn unsere Ausgangszahl weist nach dem Komma nun nur noch eine Null auf.

Zur Übung wollen wir noch eine andere Zahl mit Nachkommastellen ins Binärformat überführen, nämlich 0,1.

```
0,1x2 = 0,2   1. Nachkommastelle:0
0,2x2 = 0,4   2. Nachkommastelle:0
0,4x2 = 0,8   3. Nachkommastelle:0
0,8x2 = 1,6   4. Nachkommastelle:1
```

Jetzt läßt man – das habe ich beim ersten Beispiel noch nicht erwähnt – diese neue Vorkommastelle einfach weg und rechnet wieder mit den Nachkommastellen weiter:

```
0,6x2 = 1,2   5. Nachkommastelle:1
0,2x2 = 0,4   6. Nachkommastelle:0
0,4x2 = 0,8   7. Nachkommastelle:0
0,8x2 = 1,6   8. Nachkommastelle:1
0,6x2 = 1,2   9. Nachkommastelle:1
```

Das kommt Ihnen sicherlich von der 5. Verdoppelung her bekannt vor. Es zeigt sich, daß diese Rechnung nie aufgeht, weil sich eine periodische Zahl ergibt:

0,000 1100 1100 1100...

Das kann Ihnen öfters bei der Zahlenumwandlung passieren, daß ein endlicher Dezimalbruch in einen unendlichen periodischen Binärbruch übergeht. Kehren wir zurück zu unserem ersten Beispiel, 1985,125. Die ganze Umwandlung (Vorkomma- und Nachkommaanteil) führte zu:

111 1100 0001,001

Der dritte Schritt der Verwandlung von der Dezimalzahl zum Binärformat (nach 1.=Vorkommaanteil umwandeln, 2.= Nachkommaanteil umwandeln) ist das sogenannte Normalisieren. Das ist einfach das Verschieben des Kommas nach links (wie in unserem Beispiel) oder rechts, solange, bis vor dem Komma nur noch Nullen stehen und direkt hinter ihm eine 1. In Kapitel 29 haben wir gelernt, daß für jede Stelle, die das Komma nach links wandert, der Exponent um 1 höher wird. Unser Exponent ist im Moment noch Null (2^0 ist ja 1). Um also nach der Regel zu normalisieren, wird das Komma um 11 Stellen nach links verschoben. Der Exponent ist dann 11(dez) und unsere Zahl erscheint im neuen Gewand:

0.1111 1000 0010 01 E +1011

E +1011 heißt dabei Exponent, und wird im Binärformat dargestellt (1011 (bin.) ≙ 11 (dez.)). Soweit, sogut. Alles bisher unternommene hat Allgemeingültigkeit. Von nun an aber müssen wir uns spezialisieren auf den Commodore 64 (im VC 20 und einigen anderen Computern ist es aber auch so). Der Exponent kann ja – je nach dem, ob das Komma nach links oder nach rechts zum Normalisieren verschoben wurde – positiv sein (wie bei unserem Beispiel) aber auch negativ. Im Commodore 64 wird zum Exponenten die Zahl 128 addiert. Das ist dann Schritt 4, der im Beispiel zu 139 führt, womit wir schon das Exponentenbyte fertig haben:

Exponent: dez.139 bin.1000 1011 hex.8B

Hätten wir einen negativen Exponenten erhalten, zum Beispiel 20, dann stünde im Exponentenbyte nun dez.108, beziehungsweise dasselbe im Binärformat.

Der Rest unserer Zahl, also die Mantisse, wird nun Schritt 5 unterzogen. Zunächst läßt man das Komma weg. Die Binär-

zahl wird dann auf 4 Byte linksbündig aufgeteilt. In unserem Beispiel erhalten wir so:

1111 1000	0010 0100	0000 0000	0000 0000
Byte 1	Byte 2	Byte 3	Byte 4

Wie Sie sehen, werden die unbenutzten Bits mit Nullen aufgefüllt. Was nun noch nicht berücksichtigt wurde, ist das Vorzeichen der Mantisse. Es ist im Beispiel noch nicht zu erkennen, ob wir +1985,125 oder -1985,125 vorliegen haben. Das gehen wir nun im letzten Schritt (Nummer 6) an. Im Commodore 64 gibt es zwei Möglichkeiten der Speicherung von Fließkommazahlen. Für Schritt 6 muß man sich entscheiden, wo man die Zahl haben will.

In Kapitel 30 ist schon mal der FAC erwähnt worden, der Fließkomma-Akkumulator 1, welcher die Speicherstellen dez. 97 bis 102 ($61 bis $66) belegt. Ein zweiter Fließkomma-Akkumulator, AFAC oder ARG genannt, belegt die Plätze dez. 105 bis 110 ($69 bis $6E). Diese Akkumulatoren haben für die Fließkommarechnungen eine ähnliche Bedeutung wie der Akku für die 1-Byte-Rechnungen. Dort werden fast alle Ergebnisse abgelegt oder Zahlen abgerufen. Wir sehen, daß wir darin 6 Byte zur Verfügung haben. In Byte 97 liegt der Exponent in der von uns ermittelten Form. Byte 98 bis 101 sind die vier Mantissenbytes. Was ist in Byte 102? Das Vorzeichen! Bit 7 dieses Bytes ist 0, wenn eine positive, und 1 wenn eine negative Zahl vorliegt. Das galt für den FAC, wie Sie aus den Speicherstellen schon gesehen haben. Für den ARG ist das aber ganz genauso. Sehen wir uns nun in Bild 22 unsere Beispielzahl im FAC und im ARG nochmal an.

Im Bild ist auch angedeutet, daß die restlichen 7 Bit (Bits 0 bis 6) des Vorzeichenbytes keine Rolle spielen. Sie werden später direkt in diese Akkumulatoren hineinsehen und allerlei Bit-Müll darin finden. Lediglich Bit 7 ist für uns von Bedeutung.

Eigentlich ist das ja eine ganz schöne Verschwendung, von einem Byte wie diesem Vorzeichenbyte lediglich ein einziges Bit zu nutzen. Wenn eine beliebige Fließkommazahl irgendwo im Computer abgespeichert wird, dann gilt ein anderes Format, das MFLPT-Format (von Memory-FLoating PoinT). Man speichert hier nur in 5 Byte. Das Vorzeichenbyte fällt weg. Wie aber merkt sich der Computer das Vorzeichen? Das ist ganz schlau eingefädelt: Es gibt nämlich in den 5 Byte (1 Exponentenbyte + 4 Mantissenbyte) ein überflüssiges Bit. Sie werden sich sicher erstaunt fragen, wo?

Erinnern Sie sich doch bitte zurück an den Schritt 3, das Normalisieren. Dort wurde so verfahren, daß rechts vom Komma eine 1 steht. Wenn da aber immer und ganz grundsätzlich diese 1 steht, dann muß man sie sich eigentlich gar nicht mehr besonders merken. Man kann - vorausgesetzt, man berücksichtigt diese 1 im Bit 7 des ersten Mantissen-Bytes immer bei den Rechnungen - das Bit für andere Zwecke verwenden: Also als Vorzeichenbit. Taucht hier also eine 0 auf, dann liegt eine positive Zahl vor, ist es aber eine 1, dann signalisiert diese eine negative Zahl. Für das MFLPT-Format muß in unserem Beispiel also Bit 7 des ersten Mantissenbytes gelöscht werden (1985,125 ist ja nun mal positiv) und die komplette Zahl sieht im MFLPT-Format so aus:

1000 1011	0111 1000	0010 0100	0000 0000	0000 0000
Byte 1	↑ Byte 2	Byte 3	Byte 4	Byte 5
Exponent	M A N T I S S E			

Der Pfeil weist auf das Vorzeichenbit. Man spricht hier auch vom »gepackten« Format. Damit das alles nun nicht nur graue Theorie bleibt und Sie auch aus eigenem Erleben diese Zahlenformate sehen können, wollen wir hier ein kleines Testprogramm ausprobieren. Es wird Ihnen auch später noch gute Dienste leisten können, wenn Sie mal irgendwelche Zahlen in das FLPT- (also FAC oder ARG) oder ins MFLPT-Format umrechnen müssen. Zu Fuß ist das ja - wie Sie nun wissen - ganz schön haarig! Wie so oft, besteht auch dieses Programm aus einem Basic-Teil, der die Benutzerführung übernimmt und zwei kleinen Maschinenroutinen, die per USR-Vektor angesprungen werden. In diesen Assembler-Programmteilen sind zwei Interpreter-Routinen verborgen, die sehr nützlich und daher erklärenswert sind. Als Bild 23 ist das Basic-Aufrufprogramm abgedruckt.

Es fragt zunächst mal, ob der SMON eingeladen ist. Der wird nämlich aus dem Programm heraus angesprungen. Wird die Frage mit »J« beantwortet, dann zeigt sich ein kleines Menü, andernfalls ist das Programm beendet: Der SMON muß erst eingeladen werden.

Das Menü bietet 3 Optionen: Eine Zahl kann im FAC (Option 1), im ARG (Option 2) oder im MFLPT-Format ab Speicherstelle $6800 (Option 3) betrachtet werden.

Für Option 1 wird der USR-Vektor auf die Einsprungadresse des SMON gestellt und dann mittels USR-

FAC	$	61	62	63	64	65	66
	dez.	97	98	99	100	101	102
ARG	$	69	6A	6B	6C	6D	6E
	dez.	105	106	107	108	109	110
Inhalt							
Binär		1000 1011	1111 1000	0010 0100	0000 0000	0000	
Hex.		8B	F8	24	00	0000	0... ...
Dez.		139	248	36	0	0	
Byte Nr.		1	2	3	4	5	6
Erläuterung		Exponent	Mantisse				Vorzeichen

Bild 22.
So sieht die Zahl 1985,125 komplett im FAC und ARG aus

```
5 REM*** TEST FUER FLPT UND MFLPT ***                    <162>
10 POKE 52,96:POKE 56,96:CLR:PRINT CHR$(14
   7)CHR$(17)CHR$(17)                                    <215>
15 PRINT"IST DER SMON EINGELADEN?":INPUT"J
   /N";A$:IF A$="N"THEN END                              <254>
20 PRINT CHR$(17)CHR$(17)"{3SPACE}FLPT IN
   FAC"TAB(25)"1":PRINT                                  <003>
30 PRINT"{3SPACE}FLPT IN ARG"TAB(25)"2":PR
   INT                                                   <030>
40 PRINT"{3SPACE}MFLPT AB $6800"TAB(25)"3"
   :PRINT:PRINT                                          <077>
50 GET A$:IF A$<"1"OR A$>"3"THEN 50                      <211>
60 PRINT"AUSWAHL",A$:PRINT:INPUT"GEB
   EN SIE EINE ZAHL EIN";Z                               <107>
65 PRINT CHR$(147)                                       <142>
70 ON VAL(A$)GOTO 100,200,300                            <233>
100 REM***** FAC *****                                   <097>
110 POKE 785,0:POKE 786,192:REM USR-VEKTOR
    AUF SMON = $C000                                     <091>
120 A=USR(Z)                                             <205>
200 REM***** ARG *****                                   <213>
210 POKE 785,0:POKE 786,96:REM USR-VEKTOR
    AUF $6000                                            <011>
220 A=USR(Z)                                             <049>
300 REM***** MFLPT *****                                 <227>
310 POKE 785,0:POKE 786,97:REM USR-VEKTOR
    AUF $6100                                            <114>
320 A=USR(Z)                                             <150>
400 REM********************************                  <138>
410 REM NACH MELDUNG DES SMON MIT DEN                    <042>
420 REM KOMMANDOS                                        <221>
430 REM (1)    M 0061                                    <212>
440 REM (2)    M 0069                                    <231>
450 REM (3)    M 6800                                    <241>
460 REM DEN MONITOR EINSCHALTEN. DIE                     <137>
470 REM EINGEGEBENE ZAHL IST DANN ALS                    <148>
480 REM HEX-BYTES SICHTBAR.                              <135>
490 REM********************************                  <228>

Ⓒ 64'er
```

Bild 23. Testprogramm für die beiden kleinen Assembler-Routinen. Die Bedienung ist im Artikel erklärt.

die Zahl Z in den FAC übergeben. Es schaltet sich dann der SMON ein, der nun mittels des Kommandos M 0061 den Inhalt des FAC als Hex-Zahlen zeigt.

Option 2 richtet zunächst den USR-Vektor auf ein kleines Assembler-Programm ab $6000, welches den FAC-Inhalt in den ARG schiebt, dann den USR-Vektor auf den SMON richtet und schließlich auch diesen einschaltet. Auch hier wird mit dem M-Kommando dann per M 0069 der ARG-Inhalt sichtbar. Option 3 richtet den USR-Vektor auf eine Maschinenroutine, die bei $6100 beginnt. Dort wird der FAC-Inhalt nach $6800 und folgende Speicherstellen verschoben und zwar ins MFLPT-Format. Anschließend erfolgt dann wieder das Ausrichten des USR-Vektors auf den SMON, Anschalten des SMON, wo man durch M 68000C den Inhalt ansehen kann. Folgende Vorgehensweise empfehle ich Ihnen:

1. Einladen des SMON

2. Eintippen der beiden kleinen Assembler-Routinen mit Hilfe des SMON und Speichern (man kann einfach mit dem SMON-Kommando S→Programmname→,6000610A speichern).

2a. Wenn Sie die beiden Routinen schon gespeichert vorliegen haben, dann laden Sie sie jetzt ein. Jedenfalls sollten Sie nach dem Laden beider Assembler-Programme (SMON und die beiden Routinen) ein NEW eingeben, so daß alle Zeiger zurückgestellt werden.

3. Erst jetzt Laden oder Eintippen des Basic-Aufrufprogrammes.

Wenn Sie nun das Testprogramm starten und zum Beispiel unsere Zahl 1985,125 eingeben, werden Sie folgendes finden:

```
Option 1:
M0061
:0061 8B F8 24 00 00 78 00 00

Option 2:
M0069
:0069 8B F8 24 00 00 78 D4 CE

Option 3:
M6800
:6800 8B 78 24 00 00 FF FF FF
```

Die Bytes, welche zu unserer Zahl gehören, sind unterstrichen. Sie können jeweils nach RUN/STOP noch mit dem SMON-Kommando $8B (oder eine andere Sie interessierende Hexzahl) eine Ausgabe im Binär- und im Dezimalformat erreichen.

So, nun aber endlich zu den beiden Assembler-Routinen. Zur Option 2 gehört das folgende, bei $6000 beginnende Programm:

```
6000  JSR   BC0C
```

$BC0C ist die erste Interpreter-Routine, die wir uns zunutze machen. Sie schiebt den Inhalt vom FAC in den ARG. Mehr dazu später.

```
6003  LDA   #00
6005  STA   0311
6008  LDA   #C0
600A  STA   0312
```

Damit haben wir den USR-Vektor auf $C000 gestellt.

```
600D  JMP   C000
```

Das war das Einschalten des SMON. Im Grunde genommen könnten wir uns das Stellen des USR-Vektors ersparen.

Es ist aber sinnvoll – vor allem bei langen Programmen – wenn verstellte Vektoren nach Beendigung des Programmes auf einem definierten Wert stehen.

Nun noch die Routine für Option 3:

```
6100  LDX   #00
6102  LDY   #68
6104  JSR   BBD4
```

Auch das ist wieder eine Interpreter-Routine: Sie schiebt den FAC-Inhalt in einen Speicherbereich, dessen Startbyte durch die beiden Index-Register angegeben wird (X-Register für LSB, Y-Register für MSB, hier also 6800). Dabei wird die Zahl vom FLPT-Format in das MFLPT-Format umgewandelt. Das Prögrämmchen schließen wir ab mit einem Sprung zum Rest der ersten Routine:

```
6107  JMP   6003
```

Sehen Sie sich mal einige Zahlen im Fließkomma-Format an. Fast alle Operationen mit Zahlen vollführt unser Computer mit diesen Fließkommazahlen. Das ist dann beispielsweise der Grund dafür, daß aus einer Basic-Zeile wie der folgenden:

```
IF INT(X*10)=INT(ABS(X*10))THEN ...
```

auch bei positiven X-Werten (wo man mathematisch Gleichheit feststellt) manchmal die Bedingung als nicht erfüllt erkannt wird. X wird sofort als Fließkommazahl in den FAC gelegt, mit einer Fließkomma-Zehn multipliziert, der ABS-Wert wird ebenfalls per Fließkomma-Arithmetik ermittelt und so weiter. Dabei treten häufig Rundungsprobleme auf, wenn ein Zwischenergebnis mehr als 32 signifikante binäre Nachkommastellen aufweist (wie wir es ja zum Beispiel beim periodischen Binärbruch gesehen haben, der sich aus der simplen Dezimalzahl 0,1 ergibt). Das Rechnen mit Fließkommazahlen im Computer öffnet zwar einen ungeheuren Zahlenraum für unsere Anwendungen, es geht aber viel langsamer als die 2-Byte-Arithmetik. Immerhin müssen hier jedesmal 6 Byte (beziehungsweise 5 bei MFLPT) berücksichtigt werden. Ich glaube aber kaum, daß wir jemals in die Verlegenheit kommen werden, beispielsweise eine Fließkomma-Addition programmieren zu müssen. Eben weil unser C 64 fast alle Zahlenoperationen mit Fließkomma-Formaten durchführt, sind nahezu alle Eventualitäten schon als fertige abrufbare Programme im Interpreter enthalten. Wir müssen nur wissen, wie unsere Zahlen aussehen (das haben Sie nun ja gelernt) und wo und wie man sie für Operationen bereithält und wo und wie man die entsprechenden Routinen finden kann. Einen der wichtigsten Wege, unsere Zahlen ans Maschinenprogramm zu übergeben, haben Sie schon kennengelernt: Das Argument der USR-Funktion landet automatisch im FLPT-Format im FAC.

39. Die beiden ersten Interpreter-Routinen

Von nun an sollen nach und nach Interpreter-Routinen vorgestellt werden. Das ist allerdings nicht so einfach wie bei der Kernel-Sprungtabelle. Es gibt für die letzteren viele recht gut dokumentierte Listen. Für die Interpreter-Routinen ist kaum Literatur vorhanden. Will man die ähnlich erfassen wie die Kernel-Routinen, dann muß man ROM-Listings wälzen und vor allem probieren, probieren ... Falls Sie also mal einen Fehler in der Beschreibung feststellen oder Dinge, die ich leer lassen muß, weil mir dazu die Erleuchtung noch nicht gekommen ist, selbst schon kennen, dann schreiben Sie mir. Gemeinsam haben wir vielleicht die Chance, auch die letzte im Interpreter versteckte Nuß noch zu knacken!

Nun also zur ersten schon verwendeten Routine:

Name	MOVAF
Zweck	Übertragen des FAC in den ARG
Adresse	$BC0C, dez. 48140
Vorbereitung	Wert in FAC
Speicherstellen	$61-66 FAC, $69-6E ARG, $6F, $70
Register	Akku, X-Register
Stapelbedarf	4

Diese Routine ist deswegen so wichtig, weil viele Rechenoperationen, die zwei Zahlen verknüpfen, zwischen dem FAC und dem ARG abgewickelt werden. Wenn Sie unser kleines Testprogramm mal mit der Option 2 laufen lassen und hinterher nicht nur mit M0069 in den ARG, sondern auch mit M0061 in den FAC hineinsehen, dann stellen Sie fest, daß der FAC-Inhalt noch immer vorhanden ist.

Allerdings ist das nicht immer der Fall. MOVAF rundet nämlich – wenn nötig – vorher noch den FAC-Inhalt, der dann natürlich anders aussieht.

Fast noch häufiger benutzt man die zweite Interpreter-Routine:

Name	MOVMF
Zweck	Übertragung von FAC in Speicher unter Umrechnung ins MFLPT-Format
Adresse	$BBD4 dez. 48084
Vorbereitung	Wert in FAC, Zieladresse in X- und Y-Register (X = LSB, Y = MSB)
Speicherstellen	$61-66 FAC, $70, $22, $23
Register	Akku, X- und Y-Register
Stapelbedarf	4

Außer den unter »Speicherstellen« genannten sind natürlich auch noch die Zieladresse und deren vier nachfolgende Bytes in die Routine einbezogen (das MFLPT-Format besteht ja aus 5 Byte). $22/23 ist ein für die Operation verwendeter Zeiger.

MOVMF wird häufig dann verwendet, wenn Werte aus welchen Gründen auch immer, außerhalb der Fließkomma-Akkumulatoren gelagert werden müssen.

Es wird Ihnen vielleicht aufgefallen sein, daß im Gegensatz zur Beschreibung der Kernel-Routinen – die Rubrik »Fehler« fehlt. Der Grund ist, daß es keine solchen Sicherungen bei den Interpreter-Routinen gibt. Was passieren kann, ist unter bestimmten Bedingungen das Ansteuern von normalen Basic-Fehlermeldungen, die aber nicht immer den tatsächlichen Zustand wiedergeben. Wenn Ihnen mal bei der Programmierung mit Interpreter-Routinen Zweifel aufkommen, dann verfolgen Sie lieber den Programmweg mittels eines ROM-Listings und schalten Sie eigene Fehler-Routinen ein. Das war aber nur für die Fortgeschrittenen gesagt. Wir werden uns erst nach und nach dahin vortasten. Zunächst fehlen uns ja noch ein paar Assembler-Kenntnisse. Mit dem nächsten Abschnitt soll das besser werden.

40. Assembler-Befehle zum Beherrschen von Bits

Fangen wir also mit AND an. AND verknüpft den Akku-Inhalt Bit für Bit mit dem angegebenen Wert nach den Regeln der logischen UND-Verknüpfung. Die Adressiermöglichkeiten dieses Befehls sind allerlei:

AND 6000	absolut
AND FE	Zeropage absolut
AND #07	unmittelbar
AND 6000,X	absolut-X-indiziert
AND 6000,Y	absolut-Y-indiziert
AND (FA,X)	indiziert-indirekt
AND (FB),Y	indirekt-indiziert
AND FE,X	Zeropage-absolut-X-indiziert

Damit haben wir eine ganze Menge an Möglichkeiten. Erinnern Sie sich noch an die Regeln einer UND-Verknüpfung? Wenn nicht, dann sehen Sie sich nochmal die Tabelle 12 an.

Sie erkennen, daß zwei miteinander AND-verknüpfte Bits nur dann als Ergebnis 1 haben, wenn in beiden Bits der Wert 1 steht. Man kann mittels AND ganz gezielt Bits löschen. Nehmen wir mal als Beispiel an, wir wollten geshiftete Zeichen (das sind die mit den Codes größer als 128) in normale Zeichen umwandeln. Dazu bringen wir die Zeichencodes in den Akku und löschen Bit 7. Übrig bleibt dann der Code für das ungeshiftete Zeichen. Für das Löschen von Bit 7 brauchen wir eine sogenannte UND-Maske, die dafür sorgt, daß alle anderen Bits unverändert bleiben. An den Stellen muß in dieser Maske also eine 1 stehen (denn 0 AND 1 ergibt 0, 1 AND 1 ergibt 1). Lediglich Bit 7 der Maske muß 0 sein. Die Maske muß also heißen:

0111 1111 $7F dez. 127

Nehmen wir an, im Akku befände sich der Code für ein geshiftetes A, also dez. 193 (binär 11000001, $C1), dann ergibt die AND-Verknüpfung mit der Maske:

Akku	1100	0001	Shift A
Maske	0111	1111	
AND			
Jetzt im Akku	0100	0001	

Normales A (Code dez. 65, $41)

Man kann also, je nach Wahl der Maske, beliebige Bits löschen.

AND ist, je nach der gewählten Adressierungsart, ein 2- oder 3-Byte-Befehl. Weil das Ergebnis im Akku steht, können Flaggen beeinflußt werden. Die N- und die Z-Flagge reagieren auf das Ergebnis.

Im Gegensatz zu Basic, wo es nur eine ODER-Verknüpfung gibt, nämlich OR, existieren im Assembler zwei davon. Man unterscheidet ein »inklusives« und ein »exklusives« ODER. Die inklusive ODER-Verknüpfung des Akku mit den angegebenen Daten geschieht mit dem Assembler-Befehl ORA. ORA entspricht dem Basic-Befehl OR. Alle Adressierungsarten, die dem AND-Befehl offenstehen, können auch auf ORA angewendet werden. Wenn man Bits ORA-verknüpft, findet man folgende Ergebnisse:

0 ORA	0 = 0
0 ORA	1 = 1
1 ORA	0 = 1
1 ORA	1 = 1

Auch hier ist eine sogenannte Wahrheitstabelle recht einprägsam (siehe Tabelle 13).

Während man mit AND gezielt Bits löschen kann, ist es mit ORA möglich, Bits zu setzen. Auch dazu verwendet man eine Maske, die an allen Stellen, an denen Bits unverändert bleiben sollen, eine 0, sonst aber eine 1 enthält. Nehmen wir nochmal das Beispiel von vorhin und wandeln nun das ungeshiftete Zeichen in ein geshiftetes um. Wir müssen also Bit 7 wieder setzen: Da muß in der Maske dann eine 1 stehen. Alle anderen Bits bleiben unverändert, wenn die Maske dort eine Null aufweist. Die Maske muß daher heißen:

1000 0000 $80 dez. 128

Im Akku soll das ungeshiftete B stehen (Code dez. 66, $42, bin. 0100 0010). Die Rechnung sieht dann so aus:

Akku	0100	0010	Code für B
Maske	1000	0000	
ORA			
Jetzt im Akku	1100	0010	

AND	0	1
0	0	0
1	0	1

Tabelle 12. Wahrheitstabelle zur AND-Verknüpfung

ORA	0	1
0	0	1
1	1	1

Tabelle 13. Wahrheitstabelle zur ORA-Verknüpfung

Code für geshiftetes B.

Je nach Art der Maske kann man also ein oder mehrere Bits setzen. Im Beispiel ist auch der Einfluß dieses Befehls auf die Flaggen zu erkennen. Der Akku-Inhalt vor der ORA-Operation hatte kein Bit 7, also keine gesetzte N-Flagge. Danach ist Bit 7 gesetzt und die N-Flagge zeigt eine 1. Außer der N-Flagge kann – ebenso wie beim AND-Befehl – auch noch die Z-Flagge reagieren. ORA ist je nach Adressierungsart ein 2- oder 3-Byte-Befehl.

Während zwei Bit in der ORA-Verknüpfung eine 1 ergeben, wenn sie beide gesetzt sind oder eines von beiden, schließt die EOR-Verknüpfung den ersten Fall aus. EOR ist die exklusive ODER-Verknüpfung. Sie läßt sich sprachlich erfassen im »entweder ... oder ...«, also beispielsweise: Beim Roulette fällt die Kugel entweder auf Rouge oder auf Noir, beides zusammen ist nicht möglich. Die Regeln bei EOR sind also:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

Eine Wahrheitstabelle dazu sehen Sie in Tabelle 14.

Wozu verwendet man EOR? Es fällt Ihnen vielleicht auf, daß wir die aus Basic bekannte NOT-Funktion nicht in Assembler vorliegen haben. Obwohl EOR einige viel weitergehendere Verwendungsmöglichkeiten aufweist als NOT (aber auf Boolesche Algebra wollen wir hier nicht eingehen), kann man es mit gleicher Wirkung einsetzen. Wir haben beispielsweise in den ersten Kapiteln negative Zahlen durch Komplementieren erzeugt. Dabei sollte jedes Bit in sein Gegenteil verkehrt werden. Das wäre die Aufgabe einer NOT-Funktion. Durch ein EOR FF können wir dasselbe erreichen. Sehen wir uns wieder ein Beispiel an. Im Akku steht dez. 15 ($0F, bin. 0000 1111):

Akku	0000	1111	
Maske	1111	1111	=$FF
EOR			
Jetzt im Akku	1111	0000	

Einerkomplement von dez. 15.

Auch EOR kann alle Adressierungsarten verkraften, die die beiden anderen logischen Assembler-Befehle erlauben. Je nach der gewählten Art liegt dann ein 2- oder 3-Byte-Befehl vor. Auch hier werden die Z- und die N-Flagge beeinflußt.

Das waren also die logischen Befehle. Leider ist hier nicht der geeignete Ort, die Vielseitigkeit, die damit möglich ist, deutlich zu machen. Wenn Sie sich dafür interessieren, sollten Sie mal etwas über Boolesche Algebra lesen oder eine Einführung in die mathematische Logik.

Um dieses Thema abzuschließen, soll noch erwähnt werden, daß der Basic-Interpreter so eingerichtet ist, daß er immer dann, wenn die Richtigkeit einer Aussage zu überprüfen ist, mit –1 antwortet bei wahrer Aussage, dagegen mit 0 bei falscher. Auf diese Weise kommen diese merkwürdigen Basic-Programmzeilen ins rechte Licht, in denen Sequenzen auftauchen wie:

C=A–161–33*(A< 255)–64*(A < 192)–32*(A <160)+ 32*(A < 96)–64*(A < 64).

Jedesmal, wenn zum Beispiel A < 64 ist, tritt anstelle der Klammer ein –1 auf. Übrigens ist diese Formel eine schöne kurze Möglichkeit, ASCII-Code (hier A als Variable) in den Bildschirmcode umzurechnen (der Bildschirmcode steht dann in der Variablen C).

EOR	0	1
0	0	1
1	1	0

Tabelle 14. Wahrheitstabelle zur EOR-Verknüpfung

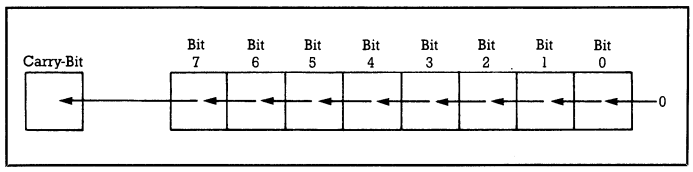


Bild 24. Wirkung des ASL-Befehls: Arithmetisches Linksschieben

Kommen wir nun zur zweiten Gruppe von Assembler-Befehlen, die Bit-Manipulationen erlauben: den Verschiebe-Befehlen. Fangen wir dabei mit ASL an, was vom englischen »Arithmetic Shift Left« kommt. Zu deutsch heißt das dann »arithmetisches nach links schieben«. Davon sind wir aber auch noch nicht schlauer. Sehen wir uns an, was dieser Befehl tut (Bild 24).

Der gesamte Inhalt des Akku beziehungsweise der Speicherstelle (je nach Adressierung) wird um eine Bit-Position nach links verschoben. Das vorherige Bit 7 wandert in die Carry-Flagge, alle anderen Bits erhalten eine um 1 höhere Position, das freigewordene Bit 0 wird mit einer 0 aufgefüllt. Toll! Aber was soll das? Zur Erklärung machen wir nochmal einen kurzen Ausflug zu unserem normalen dezimalen Zahlensystem. Nehmen wir mal die Zahl 123. Bei der Einführung in die Fließkommazahlen haben wir gelernt das Komma zu verschieben. 123 ist ja dasselbe wie 123,00. Wenn wir das Komma um eine Stelle nach rechts verschieben, erhalten wir 1230,0 (dabei lassen wir jetzt mal den Exponenten außer acht, der wäre ja –1, weil 123,00 = $1230{,}0 \times 10^{-1}$). Man kann das Ganze auch andersherum sehen: Wir haben die Zahl 123 eine Stelle nach links verschoben und die freigewordene Stelle ganz rechts mit einer Null aufgefüllt. 1230,0 ist das Zehnfache von 123,00. Die Verschiebung um eine Stelle nach links hat also zur Multiplikation unserer Zahl mit der Basis unseres Zahlensystems (also 10) geführt. Eine zweimalige Linksverschiebung führt zu 12300, den 100fachen Wert unserer Ausgangszahl. Wir haben also die Zahl 123,00 mal 10 mal 10 genommen, das sind 10^{-2}. Jede Linksverschiebung erhöht unseren Ausgangswert um eine Zehnerpotenz, oder – anders ausgedrückt – erhöht den Multiplikator um eine Zehnerpotenz und deshalb natürlich auch das Ergebnis (einmal linksschieben: Multiplikator = 10 = 10^{-4}, zweimal linksschieben: Multiplikator: 100 = 10^{-2} und so weiter).

Im Binärsystem, zu dem wir nun wieder zurückkehren, ist die Zahlenbasis die Zahl 2. Einmal Linksschieben entspricht dann einer Multiplikation mit $2^{-1}=2$. Das zweimalige Linksschieben führt zur Multiplikation mit $2^{-2}=4$ und so weiter. Nehmen wir als Beispiel die Zahl 3, welche im Binärsystem 0000 0011 heißt:

1. ASL	führt zu	0000 0110	= dez.6 (2^1x3=2x3=6)
2. ASL		0000 1100	= dez. 12 (2x6= 12, 2^2x3=4x3=12)
3. ASL		0001 1000	= dez.24 (2x12=24, 2^3x3=8x3=24)
4. ASL		0011 0000	= dez. 48 (2x24=48, 2^4x3=16x3=48)
5. ASL		0110 0000	= dez. 96 (2x48=96, 2^5x3=32x3=96)
6. ASL		1100 0000	= dez. 192 (2x96=192, 2^6x3=63x3+192)

Bis jetzt landete im Carry-Bit immer eine Null. Wenn wir nun nochmal linksschieben, finden wir darin eine 1, die offensichtlich als Bit 8 unseres Ergebnisses dienen muß:

7. ASL (1) 1000 0000 = (mit Carry als Bit 8) dez. 384 (2x192=384, 2^{-7}x3= 128x3=384

Daraus folgt, daß immer dann, wenn man sich nicht hundertprozentig sicher ist, eine Abfrage des Carry-Bits erfolgen sollte, sofern man ASL zum Rechnen einsetzt (BCC beziehungsweise BCS bieten sich da an). Dazu kommen wir noch. Sehen wir uns zunächst mal an, wie ASL adressierbar ist:

ASL

ohne Adresse, der Akkuinhalt wird nach links verschoben.

Manchmal als eigene Adressierungsart bezeichnet.

```
ASL 6000     absolut
ASL FE       Zeropage-absolut
ASL 6000,X   absolut-X-indiziert
ASL FA,X     Zeropage-absolut-X-indiziert
```

Je nach Adressierung tritt ASL dann als 1-, 2- oder 3-Byte-Befehl auf. Die N-, die Z- und die Carry-Flagge werden beeinflußt. Das Ergebnis steht bei der ersten Adressierungsart (also ASL ohne Adresse) im Akku. In den anderen Fällen findet man es in der jeweiligen Speicherstelle.

Nun gut, werden Sie sagen, man kann also mittels ASL Zahlen mit 2, 4, 8, 16 32 etc. multiplizieren. Was aber, wenn man mal 40 nehmen will? Da gibt es einige Möglichkeiten, die ein bißchen den Erfindungsgeist ansprechen. Man kann ja, wenn irgendeine Zahl Z mal 40 gerechnet werden soll, dafür schreiben:

$40*Z=(32+8)*Z=32*Z+8*Z$

Schon haben wir wieder Multiplikatoren, die den Einsatz von ASL ermöglichen. Die beiden Zwischenergebnisse (als 32xZ und 8xZ) speichern wir irgendwo ab und zählen sie dann zusammen. Wenn Z zum Beispiel 3 wäre, könnte man das so programmieren:

```
6000   STA    6100
```

Dabei sollte im Akku Z also die 3 stehen, die wir nun zwischengespeichert haben.

```
6003   ASL
6004   ASL
6005   ASL
6006   ASL
6007   ASL
```

Jetzt liegt im Akku der 32fache Wert von 3, also 96 vor und wir speichern dieses Zwischenergebnis ab.

```
6008   STA    6101
600B   LDA    6100
```

Wir haben nun den Wert 3 aus dem Zwischenspeicher $6100 wieder in den Akku geholt und schieben ihn 3mal nach links um den 8fachen Wert zu erhalten.

```
600E   ASL
600F   ASL
6010   ASL
```

Nun erfolgt das Zusammenzählen beider Zwischenergebnisse. Dabei ist ja 8xZ noch im Akku.

```
6011   CLC
6012   ADC    6101
```

Damit ist die Aufgabe gelöst. Das Ergebnis steht im Akku und kann nun weiter verwendet werden.

Auf diese Weise kann man immer einen Mulitplikator in eine Zweierpotenz (2, 4, 8, 16,...) und weitere Summanden zerlegen. Dies ist allerdings eine zwar schnelle, aber doch recht eingeschränkte Art der Multiplikation. Außerdem haben Sie noch nicht erfahren, wohin man denn nun am besten mit BCC verzweigt, wenn die 8 Bit des Ergebnisses überlaufen.

Abschließend finden Sie in Tabelle 15 noch alles Wissenswerte zu den neuen Befehlen.

41. Die restlichen Bit-Verschiebe-Operationen

Da wäre zunächst einmal das Gegenstück zu ASL. Dort ging es ja um das nach links schieben. Jetzt schieben wir nach rechts. LSR heißt der dazu nötige Befehl. Das kommt von »logical shift right« und heißt zu deutsch »logisches Rechtsschieben«. Fragen Sie mich bitte nicht, weshalb »logisches«. Jedenfalls ist LSR ebenso für logische Bitprüfungen geeignet wie ASL.

Mittels LSR wird jedes Bit der adressierten Speicherstelle um einen Platz nach rechts geschoben. An die Stelle des Bit 7 tritt eine Null und Bit 0 wandert in das Carry-Bit (siehe Bild 25).

Erinnern Sie sich noch an das dezimale Linksschieben mit ASL? Wir hatten festgestellt, daß jedes Linksschieben einer Dezimalzahl einer Multiplikation mit 10 entspricht. Hier im umgekehrten Fall, also beim Rechtsschieben, muß jedes LSR einer Division durch 10 entsprechen:

```
25000   wird durch LSR    2500
 2500   zu        "        250
  250              "         25
```

und so weiter

Geht man von der Ausgangszahl (25000) aus, dann ergibt sich der erste rechts verschobene Wert durch Division mit

$10^1=10$
der 2. durch $10^2=100$
der 3. durch $10^3=1000$, etc.

Es wird also durch Potenzen der Zahlenbasis 10 geteilt. Haben wir es – wie im Computer – mit Binärzahlen zu tun, deren Basis die 2 ist, dann teilen wir mit jedem LSR durch 2. Je nachdem, wie oft hintereinander das LSR auf eine Zahl ausgeübt wird, teilt man dann durch $2^1=2$, $2^2=4$, $2^3=8$, etc.

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
AND	absolut	3	2D	45	4	N, Z
	0-page-abs	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4 *	N, Z
	abs.-Y-indiz.	3	39	57	4 *	N, Z
	indiz.-indir.	2	21	33	6	N, Z
	indir.-indiz.	2	31	49	5 *	N, Z
	0-page-X-indiz	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-abs.	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4 *	N, Z
	abs.-Y-indiz.	3	19	25	4 *	N, Z
	indiz.-indir.	2	01	01	6	N, Z
	indir.-indiz.	2	11	17	5 *	N, Z
	0-page-X-indiz	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page abs.	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4 *	N, Z
	abs.-Y-indiz.	3	59	89	4 *	N, Z
	indiz.-indir.	2	41	65	6	N, Z
	indir.-indiz.	2	51	81	5 *	N, Z
	0-page-X-indiz	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-abs.	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz	2	16	22	6	N, Z, C

* bedeutet: Bei seitenüberschreitenden Indizierungen muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 15. Alles Wissenswerte der neuen Assembler-Befehle

Das konnte man sich alles ja schon vorstellen, nachdem ASL zur Multiplikation verwendet wurde. Auch hier muß man immer das Carry-Bit abfragen, denn die Division kann ja unter Umständen nicht aufgehen, wie das folgende Beispiel der Division von 3 durch 2 zeigt:

(3) 0000 0011 ergibt durch LSR: 0000 0001 und 1 im Carry-Bit. Das Ergebnis ist schon richtig, nämlich 1. Im Carry steht der Rest dieser Division, die 1. Weil der Rest für manche Berechnungen von Bedeutung ist, muß das Carry-Bit irgendwie erfaßt werden. Wie man das erreicht, lernen wir später noch. Leider ist diese Art der Division mittels LSR nicht so einfach verwendbar wie die Multiplikation mittels ASL. Während man dort durch geschicktes Aufteilen des Faktors ASL auch bei anderen Multiplikatoren als reine Zweierpotenzen anwenden konnte, ist das hier nicht so ohne weiteres möglich. Bei Divisionen geht man deshalb lieber andere Wege. Die zeige ich Ihnen ebenfalls etwas später.

LSR kann auf die gleiche Weise adressiert werden wie ASL:

LSR	auf den Akku bezogen
LSR 6000	absolut
LSR FE	Zeropage-absolut
LSR 6000,X	absolut-X-indiziert
LSR FA,X	Zeropage-absolut-X-indiziert

Im ersten Fall steht das Ergebnis im Akku, in den anderen Fällen in der jeweils adressierten Speicherstelle. Außer der N-Flagge, die in jedem Fall 0 wird, beeinflußt LSR auch die Carry-Flagge und unter Umständen die Z-Flagge. Je nach Adressierungsart liegt LSR als 1-Byte-, 2-Byte- oder 3-Byte-Befehl vor.

Sowohl bei ASL als auch bei LSR hatten wir festgestellt, daß man herausgeschobene Bits, falls sie noch von Bedeutung sind, irgendwie aus dem Carry-Bit (dort sind sie ja gelandet) an einen sinnvollen Ort schaffen muß. Das ist natürlich möglich über eine Befehlskette, in der zunächst das Carry-Bit abgefragt wird:
zum Beispiel:

```
6000    BCC 6007
6002    LDA #01
6004    STA 8000
6007    etc.
```

Wenn das Carry-Bit frei ist, wird alles weitere übersprungen. Wenn da drin etwas aufgetaucht ist, lädt man eine 1 (die ist ja im Carry-Bit) an die benötigte Speicherstelle (hier zum Beispiel 8000). Das kostet aber einige Bytes Speicherplatz und einige Taktzeiten Rechendauer. Außerdem erschwert sich die Programmierung, wenn man eine Zahl öfter verschiebt und dann nach 8000 alle Carry-Inhalte packen will. So kompliziert brauchen wir auch gar nicht zu arbeiten, denn unsere CPU kennt zwei Befehle, die das Bit-Verschieben und das Carry-Verschieben für uns machen. Das sind:

ROL rotate left	Linksrotieren
ROR rotate right	Rechtsrotieren

Sehen wir uns zunächst mal ROL (Bild 26) an:

Wie bei ASL wandert jedes Bit um eine Position nach links. Das Bit 7 wird dabei in das Carry-Bit verschoben. In Bit 0 gelangt aber hier nicht eine 0 (wie bei ASL), sondern der Inhalt des Carry-Bit (wohlgemerkt der Inhalt, der dort war, bevor dort Bit 7 hinein geschoben wurde). Bevor wir auf den praktischen Nährwert dieses Befehls eingehen, sollen erstmal die Adressierungsmöglichkeiten aufgeführt werden:

ROL		auf den Akku bezogen
ROL	6000	absolut
ROL	FE	Zeropageabsolut
ROL	6000,X	absolut-X-indiziert
ROL	FE,X	Zeropage-absolut-X-indiziert

Je nach Adressierung kann es sich dann wieder um einen 1-Byte- bis 3-Byte-Befehl handeln. Die N-, Z- und natürlich die Carry-Flagge sind beeinflußt und das Ergebnis des Befehls ist im Akku zu finden (erste Adressierungsart) oder in der angesprochenen Speicherstelle.

Wozu das Ganze? Abgesehen von der Möglichkeit, einzelne Bits auf diese Weise ohne Verlust aus einem Byte durch das Carry-Bit herausschieben zu können, um sie Prüfungen zu unterziehen, gibt es noch die Möglichkeit, einen Überlauf bei Rechenoperationen aufzufangen. Erinnern Sie sich an Kap. 40, wo wir mittels ASL Multiplikationen durchgeführt hatten? Dort kann es unter gewissen Umständen ja leicht geschehen, daß ein Byte für das Ergebnis nicht mehr ausreicht. Wir haben in den Beispielen schon die Überlegung durchgeführt, daß man mittels BCC oder BCS prüfen sollte, ob man eine signifikante Stelle (also eine führende 1) aus dem Byte herausgeschoben hat. Ist das der Fall, dann gibt es zwei Wege:

1) Man veranlaßt den Ausdruck eines OVERFLOW ERROR, wenn nur 1-Byte-Zahlen zulässig sind, oder
2) man schaltet um auf 2-Byte-Zahlen.

Sehen wir uns das mal an dem Schritt 7 des Beispiels aus Kapitel 40 an. Dort hatten wir die Zahl 192 (binär 1100 0000) vorliegen (zum Beispiel in Speicherstelle 7000). Im Computer werden 2-Byte-Integers in der Form LSB/MSB verarbeitet. Wir schaffen also die Speicherstelle für das MSB von 192 in 7001. Jetzt muß dort noch 0 drin stehen. Um bei nochmaliger Multiplikation mit 2 eine 16-Bit-Zahl als Ergebnis zu erhalten, verfährt man wie folgt:

```
6000 ASL 7000   Damit ist die führende 1 ins Carry-
                Bit gewandert
6003 BCC 6008   Das setzt man natürlich nur dann
                ein, wenn man nicht genau weiß,
                welches Ergebnis zu erwarten ist.
                Wenn keine 1 ins Carry-Bit
                gelangte, kann man die nächste
                Zeile überspringen.
6005 ROL 7001   Damit wurde der Inhalt des Carry-
                Bit als Bit 0 ins MSB unseres
                Ergebnisses geschoben.
6008 etc.
```

Die Funktion dieser Befehlssequenz können Sie aus Bild 27 entnehmen.

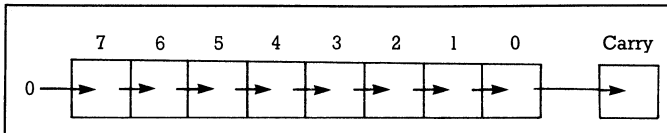


Bild 25. Wirkung von LSR auf ein Byte

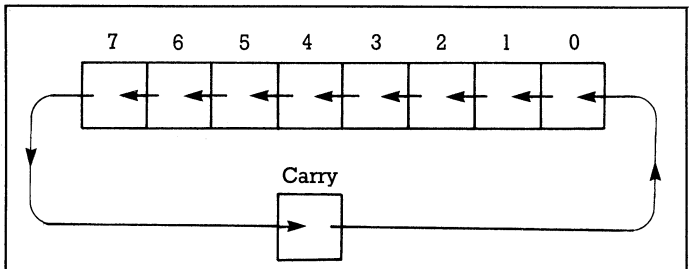


Bild 26. Wirkung von ROL auf ein Byte

Diesem Befehl werden wir später bei der 16-Bit-Multiplikation und Division noch häufig begegnen.

Sehen wir uns nun noch den letzten der Bit-Verschiebebefehle an: ROR. In Bild 28 ist schematisch gezeigt, wie rotiert wird.

Jedes Bit wandert, wie bei LSR, um eine Stelle nach rechts. Als Bit 7 kommt (im Gegensatz zu LSR) der Inhalt des Carry-Bit herein. Bit 0 wird ins Carry-Bit geschoben. Adressiert werden kann ROR ebenso wie ROL:

ROR		auf den Akku bezogen
ROR	6000	absolut
ROR	FE	Zeropage-absolut
ROR	6000,X	absolut-X-indiziert
ROR	FE,X	Zeropage-absolut-X-indiziert

Auch für die Byteanzahl, den Ort des Ergebnisses und die Flaggenbeanspruchung gilt dasselbe wie für ROL.

Die Einsatzmöglichkeiten für ROR sind allerdings geringer. Bei 16-Bit-Divisionen kann man zwar ROR einsetzen, um einen Unterlauf des MSB ins LSB aufzufangen. Weil man aber meist ohnehin andere Divisionsverfahren verwendet als das oben gezeigte mit LSR, erübrigt sich diese Anwendung in den meisten Fällen. Gut kann man ROR zu Bitprüfungen einsetzen. Das soll im nächsten Abschnitt an einem kleinen Beispiel gezeigt werden.

Zuvor aber noch eine Bemerkung: Wir sind nun durch den Befehlssatz des 6502-Assemblers fast hindurchgedrungen. Es fehlen uns nur noch – wenn ich mich nicht versehen habe – vier Befehle. Die allerdings hängen eng mit dem sogenannten Interrupthandling zusammen, das uns wohl einige Zeit beschäftigen wird.

42. Schneller Joystick

Vor einiger Zeit (64'er, Ausgabe 2/85) veröffentlichte P. Siepen eine Routine zur Abfrage des Joystickports, die eine interessante Leserbrief-Reaktion hervorrief. M. Hartig sandte nämlich einen Verbesserungsvorschlag, in dem der uns interessierende Befehl ROR die Hauptrolle spielt. Bevor ich die allerdings vorstelle, muß erst noch geklärt werden, was und wie abgefragt wird.

Wenn keine dieser Möglichkeiten angesprochen ist, enthalten diese Bits den Wert 1. Drückt man beispielsweise den Feuerknopf, dann wechselt der Inhalt von Bit 4 zum Wert 0. Man muß also ständig diese Bits überprüfen und reagieren, sobald eines davon 0 wird. Die Lösung von P. Siepen, diese Abfrage in das Interruptprogramm einzubauen, ist sehr brauchbar. Dadurch hat der Computer die Möglichkeit, trotzdem an anderen Aufgaben weiterzuarbeiten. Wir werden in den nächsten Folgen auf diese Programmiertechnik eingehen. Die Verbesserung von M. Hartig besteht darin, daß er nicht durch CMP-Befehle den Inhalt von DC00 prüft (was Zeit und auch Speicherplatz kostet), sondern mittels ROR Bit für Bit nach rechts in das Carry-Bit schiebt und dieses dann mit BCC abfragt. Sobald die Carry-Flagge nämlich frei ist, ist die zu dem Bit gehörige Joystickfunktion gefragt.

Nun die Abfrageroutine:

LDA DC00	Inhalt des DATA-Port A in den Akku
ROR	Durch Rechtsrotieren wird Bit 0 in die Carry-Flagge geschoben.
BCC Oben	Wenn die Carry-Flagge nicht gesetzt ist, war Bit 0 eine Null, also die Joystickfunktion »Oben« gefordert, zu deren Bearbeitung hier verzweigt werden kann.
ROR	Das nächste Rechtsrotieren schiebt Bit 1 in die Carry-Flagge.
BCC Unten	Auch hier wieder Abzweigen zur Bearbeitung von »Unten«, wenn Bit 1 nicht gesetzt war.
ROR	Bit 2 ins Carry-Bit
BCC Links	und bearbeiten, wenn nicht gesetzt
ROR	Bit 3 in Carry-Flagge
BCC Rechts	und verzweigen wenn Bit 3 Null war
ROR	zu guter Letzt kommt noch Bit 4 ins Carry-Bit
BCC Fire	und kann bearbeitet werden, wenn es Null war.
... weitere Bearbeitung, wenn keine Joystickfunktion	

Der Vorteil dieser nur 18 Byte langen Unterroutine liegt in ihrer Schnelligkeit: Sie braucht nur 24 Taktzyklen, wenn nicht

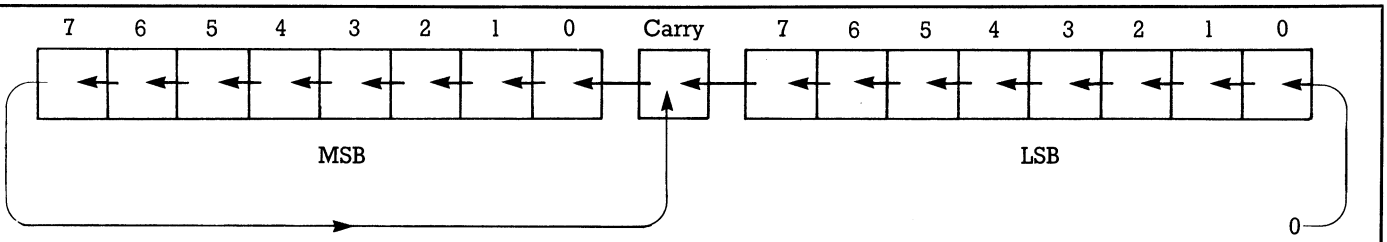


Bild 27. Wirkung der Befehlskombination ASL(-) und ROL(-) auf 2 Byte (LSB/MB)

Signale vom Joystick landen in den DATA-Ports A oder B des CIA 1. CIA heißt »Complex Interface Adapter« und ist die Institution unseres Computers, die den Verkehr mit der Außenwelt erlaubt. Wir haben zwei Stück davon (CIA 1 und CIA 2). Je nachdem, in welchen Port der Joystick gesteckt wurde, laufen die Signale in den Registern DC00 oder DC01 (dezimal 56320 oder 56321) ein. Wir nehmen im weiteren mal DC00 an. Die Bits 0 bis 4 beziehen sich auf den Joystick:

Bit 0	oben
Bit 1	unten
Bit 2	links
Bit 3	rechts
Bit 4	Feuerknopf

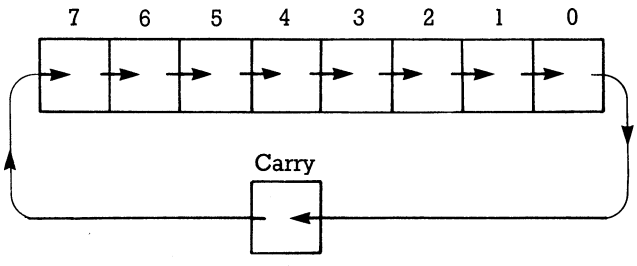


Bild 28. Wirkung von ROR auf ein Byte

verzweigt wird, beziehungsweise 25, wenn verzweigt wird. Natürlich wäre anstelle von ROR auch die Verwendung von LSR möglich gewesen, denn die herausgeschobenen Bits werden nicht mehr benötigt. Im Falle, daß man nach einer solchen Abfrage wieder den Ausgangszustand des Akku oder der Speicherstelle herstellen will, muß man eine entsprechende Anzahl ROR-Anweisungen anschließen, bis Bit 0 wieder in seine Ausgangslage rotiert ist.

43. Die 16-Bit-Multiplikation

Wir haben bisher gelernt, wie man 8-Bit-Zahlen miteinander malnehmen kann um 8- oder 16-Bit-Zahlen zu erhalten. Dabei ist unbefriedigend, daß man sich über jede Zahl Gedanken machen muß, wie man sie am besten multipliziert. Was fehlt, ist ein allgemein gültiges Programm, das in der Lage ist, jede Zahlenkombination (solange es sich um 2-Byte-Integers handelt und das Ergebnis als 16-Bit-Zahl darstellbar ist) zu verarbeiten. Und da haben wir mal wieder Glück: Gut versteckt befindet sich so etwas bereits fertig in unserem Computer. Ab dez. 45900 ($B34C) liegt im Interpreter solch eine Routine und ihr Einsprungspunkt ist für uns bei dez. 45911 ($B357). Bevor wir aber detailliert darauf eingehen, soll noch das Prinzip erklärt werden, das dabei genutzt wird.

Jeden Tag rechnen Sie wahrscheinlich völlig automatisch Multiplikationsaufgaben, ohne noch Gedanken daran zu verschwenden, wieviel Schweiß das Erlernen dieser Technik früher mal gekostet hat. Könnten Sie heute noch jemandem genau erklären, warum man da was wie macht? Genau das müssen wir aber tun, damit der Binärautomat (unser C 64) multiplizieren lernt. Nehmen wir mal eine Multiplikation von 16x15:

```
16 x 15
  16
  80
 240
```

Daß wir nicht so genau wissen, was wir da tun, liegt am ziemlich komplizierten Zehnersystem. Damit das alles einfacher und überschaubarer wird, wechseln wir mal ins Binärsystem: 16 = 10000, 15 = 1111.
Die Aufgabe sieht dann so aus:

```
10000 * 1111
  10000
   10000
    10000
     10000
--------
 11110000
```

Jetzt wird schon deutlicher, was wir getan haben. Der Faktor auf der rechten Seite wurde vom MSB an Bit für Bit durchgesehen. Jedesmal, wenn wir auf eine 1 gestoßen sind (hier waren nur Einsen), haben wir den links stehenden Faktor notiert. Dabei sind wir von mal zu mal um eine Stelle nach rechts gerückt, was mit dem Stellenwert des im rechten Faktor gerade betrachteten Bits zu tun hat. Das geschah so lange, bis alle Bits des rechten Faktors durchgearbeitet waren. Die sich auf diese Weise ergebende Kolonne wird dann addiert und führt zum Ergebnis. Vergleichen Sie, 240 ist wirklich binär 1111 0000.

Genauso wie hier beschrieben, arbeitet das Multiplikationsprogramm. Ein Unterschied tritt auf, nämlich daß nicht bis zum Schluß mit der Addition gewartet, sondern jede neue Zwischenzahl sofort addiert wird. Bild 29 zeigt die Beschreibung der Interpreterroutine:

Name	UMULT
Zweck	Multiplikation zweier 16-Bit-Zahlen
Adresse	$B357 dez. 45911
Vorbereitungen	Faktor 1 in $28/29
	Faktor 2 in $71/72
Speicherstellen	$28/29,$71/72, $5D
Register	Akku, X- und Y-Register
Stapelbedarf	keiner

Bild 29. Die Interpreterroutine UMULT

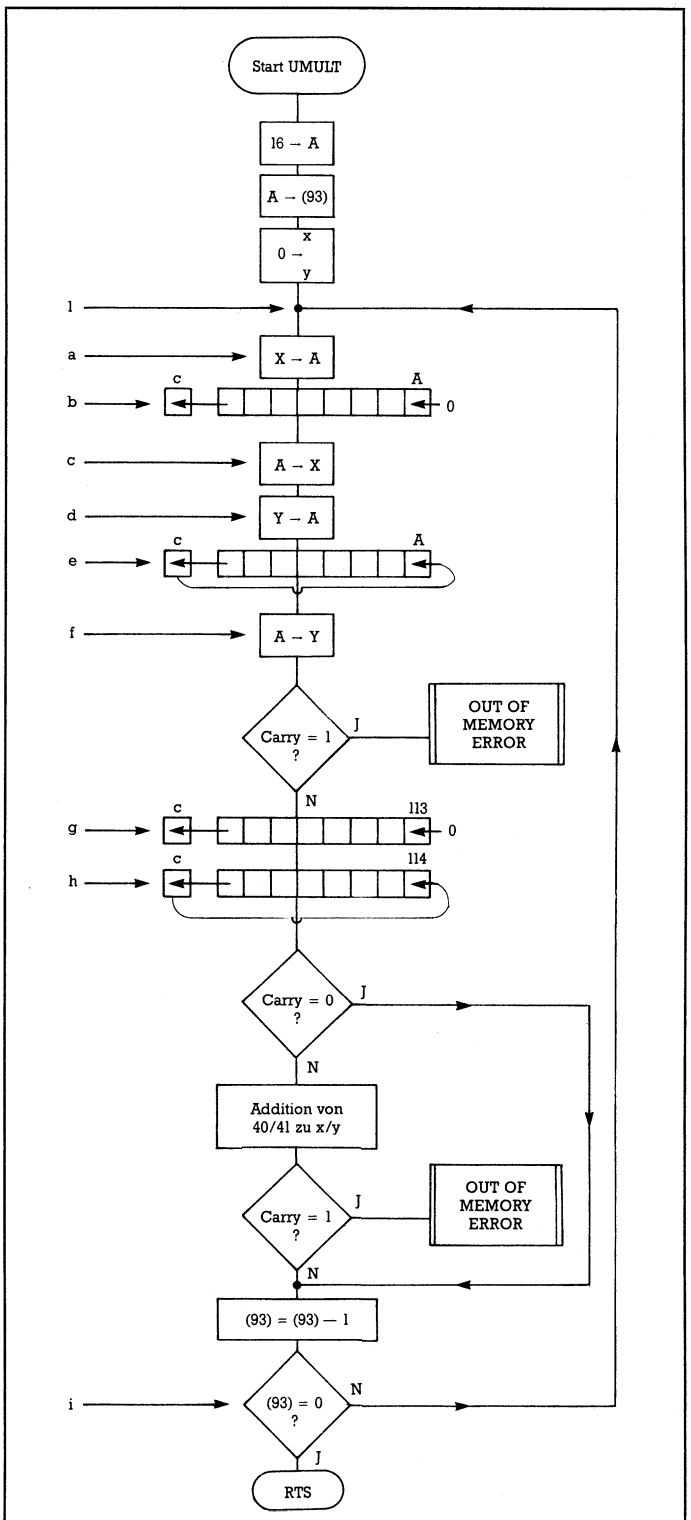


Bild 30. Flußdiagramm zur Betriebssystemroutine UMULT
Faktor 1: 40/41 Faktor 2: 113/114
Ergebnis: X/Y

	114	113	Y	X	A	93	C	
I	0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0	—	Ausgangslage
a				→	0 0 0 0 0 0 0 0		—	
b					0 0 0 0 0 0 0 0		0	
c				0 0 0 0 0 0 0 0	←		0	
d				→	0 0 0 0 0 0 0 0		0	
e					0 0 0 0 0 0 0 0		0	
f			0 0 0 0 0 0 0 0	←			0	
g		1 0 0 0 0 0 1 0					0	
h	0 0 0 0 0 0 0 0						0	Ende 1.
i	0 0 0 0 0 0 0 0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 1 1	0	Schleife
II	0 0 0 0 0 0 0 1	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 1 0	1, 0	Ende 2. Schleife
III	0 0 0 0 0 0 1 0	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 0 1	0	Ende 3. Schleife
IV	0 0 0 0 0 1 0 0	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 0 0	0	Ende 4. Schleife
V	0 0 0 0 1 0 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 1	0	Ende 5. Schleife
VI	0 0 0 1 0 0 0 0	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 0	0	Ende 6. Schleife
VII	0 0 1 0 0 0 0 0	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 1	0	Ende 7. Schleife
VIII	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0	1, 0	Ende 8. Schleife
IX	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 1 1 1	0	Ende 9. Schleife
X	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0 0 0 0 0 1 1 0	1, 0	Ende 10. Schleife
XI	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 1 0	0 0 0 0 0 1 0 1	0	Ende 11. Schleife
XII	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 1 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 1 0 0	1, 0	Ende 12. Schleife
XIII	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 0	0 0 0 0 1 0 0 0	0 0 0 0 1 0 1 0	0 0 0 0 0 0 1 1	0	Ende 13. Schleife
XIV	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 1 0 0	0 0 0 1 0 0 0 0	0 0 0 1 0 1 0 0	0 0 0 0 0 0 1 0	0	Ende 14. Schleife
XV	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 1 0 1 0 0 0	0 0 1 0 0 0 0 0	0 0 1 0 1 0 0 0	0 0 0 0 0 0 0 1	0	Ende 15. Schleife
XVIa,		0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0	0 1 0 0 0 0 0 0	0 1 0 1 0 0 0 0		0	
b,	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 1	1 0 0 0 0 0 0 1	0 1 0 1 0 0 0 1	0 0 0 0 0 0 0 0	1, 0	Ende 16. Schleife

Bild 31. UMULT am Beispiel der Multiplikation 321x65=20865

Diese Routine hier abzudrucken, wäre reine Platzverschwendung. Schalten Sie einfach den SMON ein und verlangen Sie von ihm ein Disassemblerlisting ab B357. Dort haben Sie dann für die weitere Besprechung alles parat. In Bild 30 finden Sie noch ein Flußdiagramm der UMULT-Routine.

Das Ergebnis der Multiplikation befindet sich in LSB/MSB-Form in den X/Y-Registern. Programm und Flußdiagramm wollen wir an einem Beispiel nachspielen. Dazu sollen die beiden Zahlen 321 und 65 (binär 0000 0001 0100 0001 und 0100 0001) miteinander multipliziert werden, was bekanntlich 20865 (binär 0101 0001 1000 0001) ergibt. Was Ihnen im Bild 31 als undurchdringlicher Bit-Dschungel entgegenstrahlt, ist das schrittweise Verfolgen des Programms im Computerformat, also binär.

In Bild 31 sind die Speicheradressen alle dezimal angegeben. Dort finden Sie zunächst die Ausgangslage. In Speicherstelle 40/41 steht die ganze Operation über unverändert die Zahl 321. In 113/114 finden Sie (wegen des LSB/MSB-Formates umgedreht als 114/113) unseren Faktor 65. Akku und Speicherstelle 93 stehen auf 16, dem Bitzähler. In das X- und Y-Register wurde eine Null eingelesen. Im Flußdiagramm ist diese Situation mit einer 1 gekennzeichnet. Ganz unten im Diagramm sehen Sie, daß der Bitzähler 93 erniedrigt und danach geprüft wird, ob er schon gleich Null sei. Daraus folgt, daß die große Schleife 16mal durchlaufen wird. Den ersten Durchlauf (gekennzeichnet durch kleine Buchstaben) verfolgen wir im einzelnen.

a) X-Register wird zur Bearbeitung in den Akku geschoben.

b) Mittels ASL wird das Bit 7 in die Carry-Flagge geschoben, was einen Carry-Inhalt von 0 bewirkt.

c) Der solchermaßen bearbeitete Akku-Inhalt (der sich hier nicht weiter verändert hat) geht wieder zurück ins X-Register.

d) Nun ist das Y-Register zur Bearbeitung dran. Es gelangt in den Akku.

e) Mittels ROL wandert nun das MSB des X-Registers aus dem Carry-Bit in die 0-Bit-Position des Akku

f) und alles zusammen wieder ins Y-Register. Insgesamt wird dadurch die 16-Bit-Zahl im X/Y-Register um eine Stellenzahl erhöht, was der Vorbereitung zur Addition dient. (Erinnern Sie sich bitte: Die Kolonne der Einzelergebnisse wird ja addiert). Im Diagramm (ohne Buchstabenkennzeichnung) schließt sich hier noch eine Prüfung auf einen eventuellen Überlauf an, der dann mit einer Fehlermeldung beantwortet wird.

g) Nun wird das MSB der Speicherstelle 113 nach links ins Carry geschoben. Das ist auch hier noch eine Null.

h) Anschließend wandert dieser Carry-Inhalt als Bit 0 in Speicherstelle 114. Bit 7 von 114 landet dafür im Carry. Auch hier wird auf diese Weise die ganze 16-Bit-Zahl 113/114 um ein Bit nach links geschoben und im nächsten Schritt – im Flußdiagramm wieder ohne Buchstabe – geprüft, ob da eine 1 oder eine 0 ins Carry-Bit geshiftet wurde. Wenn lediglich eine Null auftrat – wie hier –, dann springt das Programm sofort zum Herabzählen des Bitzählers 93. Tritt aber eine 1 auf, dann addiert sich der Inhalt von 40/41 zu X/Y.

i) Hier wird der Zustand der betroffenen Speicherstellen und Register nach dem ersten Schleifendurchlauf gezeigt.

Römisch II bis XVI zeigen nun jeweils den Zustand nach dem 2. bis 16. Durcharbeiten der großen Schleife.

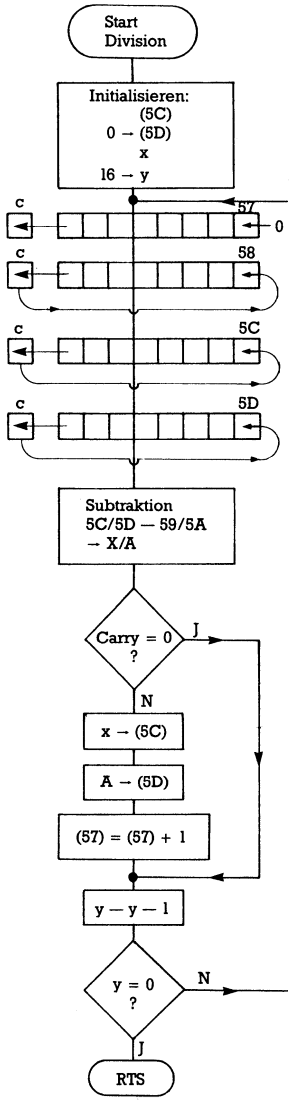


Bild 32. Flußdiagramm des Programms zur 16-Bit-Division

A	:	B	=	C	+	Rest
57/58		59/5a		57/58		5c/5d

```
.
,5000  A2 00     LDX #00
,5002  86 5C     STX   5C
,5004  86 5D     STX   5D
,5006  A0 10     LDY #10
,5008  06 57     ASL   57
,500A  26 58     ROL   58
,500C  26 5C     ROL   5C
,500E  26 5D     ROL   5D
,5010  38        SEC
,5011  A5 5C     LDA   5C
,5013  E5 59     SBC   59
,5015  AA        TAX
,5016  A5 5D     LDA   5D
,5018  E5 5A     SBC   5A
,501A  90 06     BCC 5022
,501C  86 5C     STX   5C
,501E  85 5D     STA   5D
,5020  E6 57     INC   57
,5022  88        DEY
,5023  D0 E3     BNE 5008
,5025  60        RTS
-------------------------
.?
```

Programm 1. Die 16-Bit-Division

	58	57	5A	59	5D	5C	A	X	Y	C	
I a	01010001	10000011	00000001	01000001	00000000	00000000	–	00000000	00010000	–	Ausgangslage n. Init.
b		00000110								1	1. Linksschieben
c	10100011									0	2. Linksschieben
d, e					00000000	00000000				0	3./4. Linksschieben
f	10100011	00000110	00000001	01000001	00000000	00000000	11111110	10111111	00001111	1, 0	Ende der 1. Schleife
II	01000110	00001100	00000001	01000001	00000000	00000001	11111110	11000000	00001110	1, 0, 1, 0	Ende der 2. Schleife
III	10001100	00011000	00000001	01000001	00000000	00000010	11111110	11000001	00001101	1, 0	Ende der 3. Schleife
IV	00011000	00110000	00000001	01000001	00000000	00000101	11111110	11000100	00001100	1, 0, 1, 0	Ende der 4. Schleife
V	00110000	01100000	00000001	01000001	00000000	00001010	11111110	11001001	00001011	1, 0	Ende der 5. Schleife
VI	01100000	11000000	00000001	01000001	00000000	00010100	11111110	11010011	00001010	1, 0	Ende der 6. Schleife
VII	11000001	10000000	00000001	01000001	00000000	00101000	11111110	11100111	00001001	1, 0, 1, 0	Ende der 7. Schleife
VIII	10000011	00000000	00000001	01000001	00000000	01010001	11111111	00010000	00001000	1,1,0,1,0	Ende der 8. Schleife
IX	00000110	00000000	00000001	01000001	00000000	10100011	11111111	01100010	00000111	1, 0, 1, 0	Ende der 9. Schleife
X a	00001100	00000000			00000001	01000110	00000000	00000101		1, 0, 1, 1	
b	00001100	00000001	00000001	01000001	00000000	00001101	00000000	00000101	00000110		Ende der 10. Schleife
XI	00011000	00000010	00000001	01000001	00000000	00001010	11111110	11001001	00001010	1, 0	Ende der 11. Schleife
XII	00110000	00000100	00000001	01000001	00000000	00010100	11111110	11010011	00001100	1, 0	Ende der 12. Schleife
XIII	01100000	00001000	00000001	01000001	00000000	00101000	11111110	11100111	00000011	1, 0	Ende der 13. Schleife
XIV	11000000	00010000	00000001	01000001	00000000	01010000	11111111	00001111	00000010	1, 0	Ende der 14. Schleife
XV	10000000	00100000	00000001	01000001	00000000	10100001	11111111	01100000	00000001	1, 0, 1, 0	Ende der 15. Schleife
XVIa,	00000000	01000000			00000001	00000011	00000000	00000010		1,1,0,1,1	
b,	00000000	01000001	00000001	01000001	00000000	00000010	00000000	00000010	00000000		Ende der 16. Schleife = Endlage

Bild 33. 16-Bit-Division Schritt für Schritt am Beispiel 20867:3

Wenn Sie verstehen möchten, was da passiert, sollten Sie versuchen, Bild 31 nur als Kontrolle zu verwenden und ansonsten mal selbst alle Schritte nachzuvollziehen.

44. 16-Bit-Division

Beim umgekehrten Weg, nämlich der Teilung von zwei 16-Bit-Zahlen, haben wir nicht so viel Glück: Ich konnte keine derartige Routine im Interpreter entdecken. Nun gibt es aber fast in jedem Lehrbuch der Maschinensprache die Vorstellung eines solchen Programms, so daß man sich das schönste aussuchen kann. Das Prinzip ist auch da dasselbe, wie wir es von der normalen Division gewohnt sind: Der Divisor wird Schritt für Schritt vom Dividenden abgezogen. In der Literatur [1] fand ich eine sehr kurze Routine, die ich Ihnen leicht modifiziert als Programm 1 vorstellen will.

In Bild 32 ist ein Flußdiagramm dieser Routine gezeigt und in Bild 33 lacht Ihnen wieder das Bit-Gewirr entgegen, das Sie schon von der Multiplikation her kennen, hier aber für die Division.

Damit Sie wissen, wo was hinein- oder herauskommt:

A	: B	= C	+ Rest
↑	↑	↑	↑
$57/58	$59/5A	$57/58	$5C/5D

An dem folgenden Beispiel soll der Programmverlauf getestet werden: Wir teilen 20867 durch 321. Dabei kommt nach Adam Riese heraus: 65, Rest 2.

In folgender Weise wird in die Speicherzellen die Aufgabe eingespeist:

20867	$57	1000	0011	LSB
	$58	0101	0001	MSB
321	$59	0100	0001	LSB
	$5A	0000	0001	MSB

Als Ergebnis findet man dann:

65	$57	0100	0001	LSB
	$58	0000	0000	MSB
Rest 2	$5C	0000	0010	LSB
	$5D	0000	0000	MSB

Als Bit-Zähler dient hier das Y-Register.

b) Erstes Linksschieben des LSB mittels ASL. Dabei gelangt die 1 in das Carry-Bit.

c) Hineinrotieren der 1 aus dem Carry in das MSB mittels ROL.

d), e) Linksrotieren der 16-Bit-Zahl in $5C/5D, die jetzt noch 0 ist.

f) Situation am Ende der ersten Schleife. Der Bitzähler ist um 1 reduziert.

Im folgenden wird dann jeweils die Situation am Ende der Schleife gezeigt. Beim Berechnen der Differenz muß jeweils darauf geachtet werden, daß die Subtraktion einer Zahl als Addition des Zweierkomplements ausgeführt wird. Das haben wir in Kap. 11 und 14 kennengelernt. Allerdings muß an dieser Stelle nochmal gesagt werden, daß die 1, die zum Einerkomplement hinzuaddiert wird, um das Zweierkomplement zu erhalten, das gesetzte Carry-Bit ist. Nun dürfte es für Sie eigentlich keine Probleme mehr geben, was das Nachvollziehen der Divisionsroutine betrifft.

Damit dürfen wir getrost die 16-Bit-Arithmetik abschließen. Alle vier Grundrechnungsarten können Sie jetzt programmieren. Weitere Rechenarten, wie Potenzieren, das Ziehen von Wurzeln, Logarithmen etc. bedingen ohnehin, daß die Argumente oder Ergebnisse keine Integerzahlen sind. Hier werden wir dann mit Fließkommaarithmetik arbeiten und den dazu vorgesehenen Interpreterroutinen.

```
 1 REM ********************************    <250>
 2 REM *                              *    <229>
 3 REM *          PROGRAMM  2         *    <125>
 4 REM *                              *    <231>
 5 REM *   ERSTELLEN UND AUFRUF EINES  *    <186>
 6 REM *        HILFSBILDSCHIRMES     *    <216>
 7 REM *                              *    <234>
 8 REM * HEIMO PONNATH    HAMBURG 1985 *    <082>
 9 REM ********************************    <002>
10 PRINT CHR$(147):POKE 785,0:POKE 786,96:
   GOTO 30                                 <095>
15 REM -------- UP CURSOR SETZEN ------     <112>
20 POKE 211,SP:POKE 214,Z:SYS 58640:RETURN <163>
25 REM- ERSTELLEN DES HILFSBILDSCHIRMES-    <123>
30 Z=1:SP=1:GOSUB 20:PRINT"***************
   ************************"               <151>
40 Z=21:SP=1:GOSUB 20:PRINT"**************
   ************************"               <211>
50 Z=10:SP=7:GOSUB 20:PRINT"TEST FUER DIE
   VERSCHIEBUNG"                           <110>
55 REM ---- AUFRUF ZUM VERSCHIEBEN ----    <033>
60 A=USR(DUMMY)                            <195>
65 REM ---BILDSCHIRM NEU BESCHREIBEN---    <193>
70 GET A$:IF A$=""THEN 70                  <122>
80 PRINT CHR$(147):Z=2:SP=2:GOSUB 20:PRINT
   "JETZT SOLLTE DER ALTE BILDSCHIRM"      <092>
90 Z=4:SP=2:GOSUB 20:PRINT"UNTER DAS KERNA
   L-ROM GESCHOBEN SEIN"                   <150>
100 PRINT:PRINT:PRINT" -- JEDER{2SPACE}USR
    -AUFRUF HOLT DEN --"                   <003>
110 PRINT" -- HILFSBILDSCHIRM WIEDER .{3SP
    ACE}--"                                <068>
120 PRINT" -- AUCH IM DIREKT-MODUS{7SPACE}
    --"                                    <056>
130 PRINT:PRINT:PRINT"{2SPACE}PROBIEREN SI
    E MAL: A=USR(1) [RETURN]"              <050>
140 Z=19:SP=0:GOSUB 20:END                 <164>
```

© 64'er

Programm 2. Das Demo-Programm zur neuen Verschieberoutine. Vorher müssen Programm 3 und Programm 4 geladen werden.

45. Das Programmprojekt wird fortgeführt

Im Kap. 32 haben wir ein Projekt gestartet, das dort eine Kopfzeile rückholbar unter den oberen ROM-Bereich verschob. Unser Wissen ist seither gestiegen und damit auch unsere Ansprüche. Eine Kopfzeile reicht nicht mehr, jetzt soll es ein ganzer Hilfsbildschirm sein, den wir erst in aller Ruhe erstellen wollen, um ihn dann jederzeit abrufbar unter das Betriebssystem zu packen. Den Aufruf wollen wir wieder mit der USR-Funktion steuern. Diesmal soll aber so programmiert werden, daß der Hilfsbildschirm erhalten bleibt und man ihn also mehrfach einblenden kann. Über die Nützlichkeit einer solchen Routine braucht man sicherlich nicht viele Worte zu verlieren: Denken Sie da nur mal an Programme, die irgendwelche Tasten mit besonderen Funktionen belegen, für die Sie eine Gedächtnisstütze brauchen, oder...

Als Programm 2 ist ein kleines Demo-Programm abgedruckt, welches zuerst einen Bildschirm erstellt, dann die Routine »Verschieben« aufruft, den Bildschirm löscht und neu beschreibt und schließlich mit einem weiteren USR den alten Bildschirm einblendet (vorher Programm 3 und 4 laden).

Von nun an können Sie immer – auch im Direktmodus – durch ein USR-Kommando diesen Bildschirm abbilden. Zum Programm in Kap. 32 sind noch zwei Dinge zu bemerken, die hier geändert werden sollen. Erstens eine Frage: Ist Ihnen der Computer mal abgestürzt beim Aufruf des Programms? Die Wahrscheinlichkeit dafür ist ungefähr 1 : 60, wenn nämlich ein Interrupt stattfindet, während die Speicherstelle 1 geändert wird. Obwohl wir erst in den nächsten Kapiteln auf Interrupts eingehen werden, wollen wir die Wahrscheinlichkeit für so einen Absturz auf Null reduzieren. Eine andere Sache ist der Ort, an dem sich das Programm befand. Es hat sich nämlich herausgestellt, daß anscheinend die Nutzung dieses dort gewählten Speicherbereichs nicht ganz so problemlos ist. Bei einigen Anrufen wurde mir erzählt, daß zumindest der Anfang ab $02A7 bei bestimmten Konstellationen überschrieben wird. Deswegen packen wir unser Programm ganz unkonventionell nach $6000, von wo Sie es – das beherrschen Sie ja mit dem SMON inzwischen sicher – dorthin schieben können, wo es Ihnen gefällt. Allerdings müssen dann auch die USR-Adressen geändert werden. Aber auch das dürfte für Sie inzwischen kein Problem mehr sein.

```
.
,6000  A9 00     LDA #00
,6002  85 5F     STA   5F
,6004  A9 04     LDA #04
,6006  85 60     STA   60
,6008  A9 E8     LDA #E8
,600A  85 5A     STA   5A
,600C  85 58     STA   58
,600E  A9 07     LDA #07
,6010  85 5B     STA   5B
,6012  A9 E3     LDA #E3
,6014  85 59     STA   59
,6016  20 BF A3  JSR A3BF
,6019  A9 00     LDA #00
,601B  85 5F     STA   5F
,601D  A9 D8     LDA #D8
,601F  85 60     STA   60
,6021  A9 E8     LDA #E8
,6023  85 5A     STA   5A
,6025  A9 DB     LDA #DB
,6027  85 5B     STA   5B
,6029  A9 D1     LDA #D1
,602B  85 58     STA   58
,602D  A9 E7     LDA #E7
,602F  85 59     STA   59
,6031  20 BF A3  JSR A3BF
,6034  A9 40     LDA #40
,6036  8D 11 03  STA 0311
,6039  A9 60     LDA #60
,603B  8D 12 03  STA 0312
,603E  60        RTS
.?
```

Programm 3. Erster Teil der Verschieberoutine

Um diese immerhin schon 2 000 Byte (1 000 für den Bildschirm und nochmal 1 000 für das Farb-RAM) zu verschieben, bedienen wir uns einer Interpreter-Routine, die seit Ausgabe 3/85 des 64'er auch beim Checksummer verwendet wird – der Blockverschiebe-Routine (Bild 34).

Name	**BLTUC**
Zweck	Verschieben von Speicherinhalten im Speicher
Adresse	$A3BF dez. 41919
Vorbereitungen	Quelle Startadresse nach $5F/60 Endadresse+1 nach $5A/5B Ziel Endadresse+1 nach $58/59
Speicherstellen	$58-5B, $5F, $60, $22
Register	Akku, X- und Y-Register
Stapelbedarf	keiner

Bild 34. BLTUC

Wieder besteht unser Programm aus zwei Teilen. Im ersten wird der aktuelle Bildschirm nach oben geschoben. Dieser Teil speist lediglich zuerst die Adressen des Bildschirms und des Betriebssystem-ROM in die Abholspeicherstellen der danach aufgerufenen Routine BLTUC und wiederholt diesen Vorgang für die Bildschirmfarbspeicheradressen. Danach verstellen wir noch den USR-Vektor und kehren mit RTS ins Basic-Programm zurück (siehe Programm 3).

Komplexer ist der zweite Teil. Um nämlich die Informationen unter dem ROM lesen zu können, muß dieses ausgeschaltet werden. Leider läßt sich das Betriebssystem-ROM nur zusammen mit dem Basic-Interpreter ausschalten. $A3BF ist aber eine Interpreter-Routine! Da bleibt uns nichts anderes übrig, als diese Routine in unser Programm einzubauen, was uns die Gelegenheit gibt, sie uns mal etwas anzusehen. Als Bild 35 ist sie im Flußdiagramm abgebildet.

Programm 4 zeigt den zweiten Teil unseres Hilfsbildschirm-Programms.

Von $6040 an, wohin wir am Ende des ersten Teils den USR-Vektor gerichtet haben, wird zunächst wieder Quell- und Zielbereich in den Abholspeicherstellen spezifiziert und jeweils danach zuerst für den Bildschirm, dann für das Farb-RAM, das übernommene Unterprogramm angesprungen. Ab $6077 liegt dann das modifizierte Unterprogramm. Die Befehle SEI und CLI gehören zu den wenigen, die Sie erst noch kennenlernen. Sie sind es, die die Absturzwahrscheinlichkeit auf Null bringen. Jedenfalls wird zuerst das ROM aus- und dafür das RAM eingeschaltet. Ab $607F bis $60B9 befindet sich die Interpreter-Routine BLTUC. Darin wird zunächst die Länge des zu verschiebenden Bereichs berechnet, dann festgestellt, ob nur ganze Pages (Seiten) oder auch ein Restbereich verschoben werden soll. Falls ein solcher Restbereich vorhanden ist, wird auch seine Länge berechnet und zuerst dieser verschoben. Daran schließt sich das Ver-

```
.
,603F  EA        NOP
,6040  A9 00     LDA #00
,6042  85 5F     STA   5F
,6044  A9 E0     LDA #E0
,6046  85 60     STA   60
,6048  A9 E8     LDA #E8
,604A  85 5A     STA   5A
,604C  85 58     STA   58
,604E  A9 E3     LDA #E3
,6050  85 5B     STA   5B
,6052  A9 07     LDA #07
,6054  85 59     STA   59
,6056  20 77 60  JSR 6077
,6059  A9 E9     LDA #E9
,605B  85 5F     STA   5F
,605D  A9 E3     LDA #E3
,605F  85 60     STA   60
,6061  A9 D1     LDA #D1
,6063  85 5A     STA   5A
,6065  A9 E7     LDA #E7
,6067  85 5B     STA   5B
,6069  A9 E8     LDA #E8
,606B  85 58     STA   58
,606D  A9 DB     LDA #DB
,606F  85 59     STA   59
,6071  20 77 60  JSR 6077
,6074  60        RTS
-----------------------------
,6075  EA        NOP
,6076  EA        NOP
,6077  78        SEI
,6078  A5 01     LDA   01
,607A  48        PHA
,607B  A9 35     LDA #35
,607D  85 01     STA   01
,607F  38        SEC
,6080  A5 5A     LDA   5A
,6082  E5 5F     SBC   5F
,6084  85 22     STA   22
,6086  A8        TAY
,6087  A5 5B     LDA   5B
,6089  E5 60     SBC   60
,608B  AA        TAX
,608C  E8        INX
,608D  98        TYA
,608E  F0 23     BEQ 60B3
,6090  A5 5A     LDA   5A
,6092  38        SEC
,6093  E5 22     SBC   22
,6095  85 5A     STA   5A
,6097  B0 03     BCS 609C
,6099  C6 5B     DEC   5B
,609B  38        SEC
,609C  A5 58     LDA   58
,609E  E5 22     SBC   22
,60A0  85 58     STA   58
,60A2  B0 08     BCS 60AC
,60A4  C6 59     DEC   59
,60A6  90 04     BCC 60AC
,60A8  B1 5A     LDA  (5A),Y
,60AA  91 58     STA  (58),Y
,60AC  88        DEY
,60AD  D0 F9     BNE 60A8
,60AF  B1 5A     LDA  (5A),Y
,60B1  91 58     STA  (58),Y
,60B3  C6 5B     DEC   5B
,60B5  C6 59     DEC   59
,60B7  CA        DEX
,60B8  D0 F2     BNE 60AC
,60BA  68        PLA
,60BB  85 01     STA   01
,60BD  58        CLI
,60BE  60        RTS
-----------------------------
.?
```

Programm 4. Zweiter Teil der Verschieberoutine

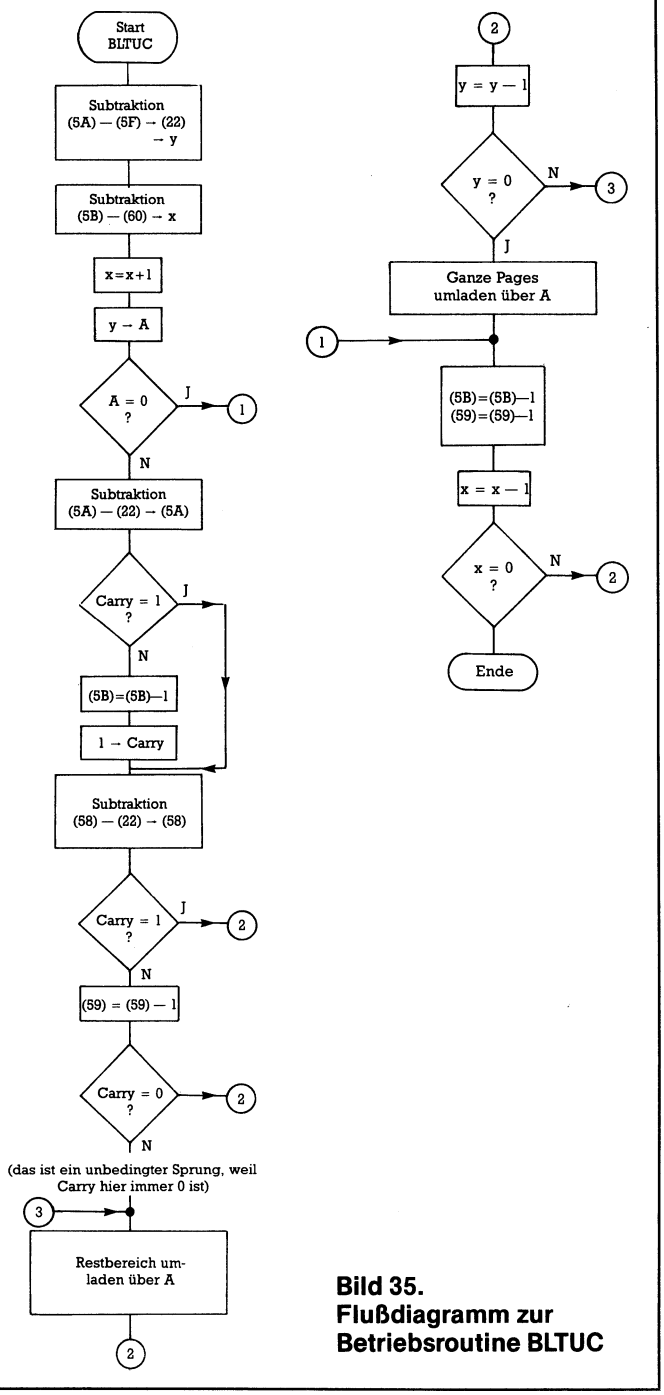


Bild 35. Flußdiagramm zur Betriebsroutine BLTUC

schieben der ganzen Pages an. Das X- und das Y-Register dienen dabei als Zähler.

Ab $60BB schließt sich wieder unsere eigene Routine an, in der wir die ROMs wieder einschalten. Auf diese Weise lassen sich noch mehrere Hilfsbildschirme unter ROM-Bereiche packen. Vielleicht überlegen Sie sich mal dazu einen Weg?

Startadresse	Format	Zahl
$AEA8	MFLPT	Pi
$B1A5	MFLPT	—32768
$B9BC	MFLPT	1
$B9C1	1-Byte-Integer	3
$B9C2	MFLPT	0.434255942
$B9C7	MFLPT	0.576584541
$B9CC	MFLPT	0.961800759
$B9D1	MFLPT	2.88539007
$B9D6	MFLPT	0.707106781 = SQR(1/2)
$B9DB	MFLPT	1.41421356 = SQR(2)
$B9E0	MFLPT	—0.5
$B9E5	MFLPT	0.693147181 = ln2
$BAF9	MFLPT	10
$BDB3	MFLPT	99999999.9
$BDB8	MFLPT	999999999
$BDBD	MFLPT	1000000000
$BF11	MFLPT	0.5
$BF16	4-Byte-Integer	—100000000
$BF1A	— " —	10000000
$BF1E	— " —	—1000000
$BF22	— " —	100000
$BF26	— " —	—10000
$BF2A	— " —	1000
$BF2E	— " —	—100
$BF32	— " —	10
$BF36	— " —	—1
$BF3A	— " —	—2160000
$BF3E	— " —	216000
$BF42	— " —	—36000
$BF46	— " —	3600
$BF4A	— " —	—600
$BF4E	— " —	60
$BFBF	MFLPT	1.44269504 = 1/1n2
$BFC4	1-Byte-Integer	7
$BFC5	MFLPT	2.14987637E-05
$BFCA	MFLPT	1.43523140E-04
$BFCF	MFLPT	1.34226348E-03
$BFD4	MFLPT	9.61401701E-03
$BFD9	MFLPT	0.0555051269
$BFDE	MFLPT	0.240226385
$BFE3	MFLPT	0.693147186 = ln2
$BFE8	MFLPT	1
$E08D	MFLPT	11879546
$E092	MFLPT	3.92767774E-08
$E2E0	MFLPT	1.57079633 = Pi/2
$E2E5	MFLPT	6.28318531 = 2*Pi
$E2EA	MFLPT	0.25
$E2EF	1-Byte-Integer	5
$E2F0	MFLPT	—14.3813907
$E2F5	MFLPT	42.0077971
$E2FA	MFLPT	—76.7041703
$E2FF	MFLPT	81.6052237
$E304	MFLPT	—41.3417021
$E309	MFLPT	6.28318531 = 2*Pi
$E33E	1-Byte-Integer	11
$E33F	MFLPT	—6.8473912E—04
$E344	MFLPT	4.85094216E—03
$E349	MFLPT	—0.0161117018
$E34E	MFLPT	0.034209638
$E353	MFLPT	—0.0542791328
$E358	MFLPT	0.0724571965
$E35D	MFLPT	—0.0898023954
$E362	MFLPT	0.110932413
$E367	MFLPT	—0.142839808
$E36C	MFLPT	0.19999912
$E371	MFLPT	—0.333333316
$E376	MFLPT	1
$E3BA	MFLPT	0.811635157
$E8DA - $E8E9	1-Byte-Integers	Tabelle der Farbcodes
$EB81 - $EBC1	— " —	Tastaturdecodierung: Einzelne Tasten
$EBC2 - $EC02	1-Byte-Integers	Tasten mit Shift
$EC03 - $EC43	1-Byte-Integers	Tasten mit Commodore-Taste
$EC78 - $ECB8	1-Byte-Integers	Tasten mit Control-Taste
$ECB9 - $ECE5	1-Byte-Integers	VIC-II-Chip-Registerwerte
$ECF0 - $ED08	1-Byte-Integers	Tabelle der LSBs der Bildschirm-Anfangsadressen

Tabelle 16. Im ROM stehen nicht nur Programme, sondern auch Tabellen, hier einige wichtige Zahlen.

46. Die ROM-Bereiche als Datenquelle

Die ROM-Bereiche enthalten nicht nur ausgeklügelte Maschinenprogramme, sondern auch eine Menge Daten. Sollten Sie mal in die Verlegenheit kommen, beispielsweise die Zahl Pi im MFLPT-Format verwenden zu müssen, dann erfordert das einen ganz schönen Aufwand an Rechen- und Programmarbeit, oder Sie möchten bestimmte Texte wie beispielsweise eine Fehlermeldung verfügbar halten ... und so weiter. Viele von diesen Daten sind schon in der Firmware enthalten und wir werden im folgenden festhalten, wo sie sich befinden und welches Format man vorfindet. Sehen wir uns zunächst Zahlen an (Tabelle 16): Es existieren noch weitere Zahlentabellen in den ROM-Bereichen, die aber selten von Interesse sind. Ebenso wie Zahlen, findet man auch Texte im ROM als ASCII—Werte abgelegt (Tabelle 17)

Sollten Sie mal in die Verlegenheit kommen, solche Texte ausgeben zu wollen, dann legen Sie sie nicht nochmal in einer eigenen Texttabelle ab, sondern schöpfen Sie aus dem Fundus, den wir im ROM-Bereich fix und fertig haben.

Nun noch die Tabelle 18 mit den neuen Assembler-Befehlen.

[1] »Computerspiele und Wissenswertes Commodore 64«, Haar bei München: Markt & Technik Verlag, 1984. Das ist die von P. Lücke besorgte Übersetzung des amerikanischen Buches »More on the sixtyfour« und ist jedem Assembler-Programmierer zu empfehlen.

$A004	CBMBASIC
$A09E - $A19D	Texte der Basic-Befehlsworte (im letzten Byte ist jeweils Bit 7 gesetzt)
$A19E - $A327	Texte der Basic-Fehler- und System-Meldungen. (Im letzten Byte ist jeweils Bit 7 gesetzt)
$A364 - $A38A	Weitere System-Meldungen: OK, ERROR, IN, READY, BREAK. (Das letzte Byte ist jeweils 0)
$ACFC - $AD1D	Fehlermeldungstexte für INPUT: ?EXTRA IGNORED, ?REDO FROM START. (Das letzte Byte ist jeweils 0)
$E460	BASIC BYTES FREE
$E473	**** COMMODORE 64 BASIC V2 **** 64K-RAM-System
$ECE6	LOAD (Return) RUN (Return)
$F0BD - $F12B	Texte für Ein- und Ausgabe-Operationen
$FD10	CBM80

Tabelle 17. Diese Texte sind im ROM als ASCII-Werte abgelegt

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
LSR	»Akkumulator«	1	1A	26	2	N,Z,C
	absolut	3	4E	78	6	N,Z,C
	0-page-absolut	2	46	70	5	N,Z,C
	absolut-X-indiz.	3	5E	94	7	N,Z,C
	0-page-X-indiz.	2	56	86	6	N,Z,C
ROL	»Akkumulator«	1	2A	42	2	N,Z,C
	absolut	3	2E	46	6	N,Z,C
	0-page-absolut	2	26	38	5	N,Z,C
	absolut-X-indiz.	3	3E	62	7	N,Z,C
	0-page-X-indiz.	2	36	54	6	N,Z,C
ROR	»Akkumulator«	1	6A	106	2	N,Z,C
	absolut	3	6E	110	6	N,Z,C
	0-page-absolut	2	66	102	5	N,Z,C
	absolut-X-indiz.	3	7E	126	7	N,Z,C
	0-page-X-indiz.	2	76	118	6	N,Z,C

Tabelle 18. Die neu besprochenen Assembler-Befehle

47. Was sind Interrupts?

Die Assembler-Befehle haben wir bis auf vier noch offenstehende alle behandelt. Diese vier, die alle mit dem Interrupt-Handling zusammenhängen, sollen nun unser Thema sein. Wenn wir sie beherrschen, haben wir den ersten Schritt zum Meister der Assembler-Alchimie getan. Diese vier kleinen 1-Byte-Befehle öffnen uns eine geheime Pforte zu einem Universum an Programmier-Möglichkeiten, von dem wir bisher kaum zu träumen vermochten. Genug der Schwärmerei, erst kommt noch eine Menge Arbeit, die uns wohl mehrere Kapitel in Atem halten wird.

Zuvor noch eine Bemerkung: es gibt kaum ein Thema im Rahmen der Programmierung in Assembler, welches so penetrant häufig Abstürze provoziert, wie das nunmehr angesteuerte! Falls Sie noch keine RESET-Taste an ihrem Computer haben, wird es nun höchste Zeit. Diese nützlichen Dinger werden inzwischen schon so preiswert angeboten (sehen Sie mal in den Kleinanzeigenteil!), daß Sie zur Grundausstattung eines Assembler Alchimisten zählen.

Unser Computer ist – solange er eingeschaltet ist – ständig mit irgendwelchen Tätigkeiten beschäftigt. Im Direktmodus hängt er beispielsweise meistens in einer Warteschleife und harrt der Eingaben, im Programm-Modus arbeitet er sich mit Hilfe der Interpreterschleife durch einen Basic-Befehlstext hindurch und so weiter. Nun werden Sie ja sicher schon festgestellt haben, daß er im Direktmodus auch den Cursor blinken läßt, in beiden Modi die TI$-Uhr weiterzählt und weitere Dinge macht, die anscheinend so nebenher passieren. Schon in Kapitel 8 aber haben wir einen Unterschied zwischen Mensch und Computer festgehalten: Der Mensch kann mehrere Dinge gleichzeitig tun, der Mikroprozessor ist nur fähig zu einer Arbeit pro Zeiteinheit. Weil aber diese Zeiteinheiten so unfaßbar kurz sind (etwa eine Millionstel Sekunde), haben wir Benutzer den Eindruck der Gleichzeitigkeit.

Wenn dem aber so ist, wie macht es der Computer, daß er beispielsweise ein Programm abarbeitet und trotzdem die TI$-Uhr weiterzählt? Durch Unterbrechungen (interrupt = unterbrechen) der gerade ausgeübten Tätigkeit. Ein Beispiel aus dem täglichen Leben soll uns das illustrieren: Sie lesen gerade diesen Artikel, da klingelt das Telefon und ein Freund möchte von Ihnen wissen, was eigentlich Unterbrechungen sind. Während Sie es ihm erklären, fängt in der Küche der Teekessel schrill zu pfeifen an. Sie sagen Ihrem Freund, er möge sich einen Moment gedulden, gehen in die Küche und nehmen den Kessel vom Feuer. Dann kehren Sie ans Telefon zurück und beenden nach einer Weile das Gespräch. Nach dem Auflegen des Telefonhörers setzen Sie die Lektüre des

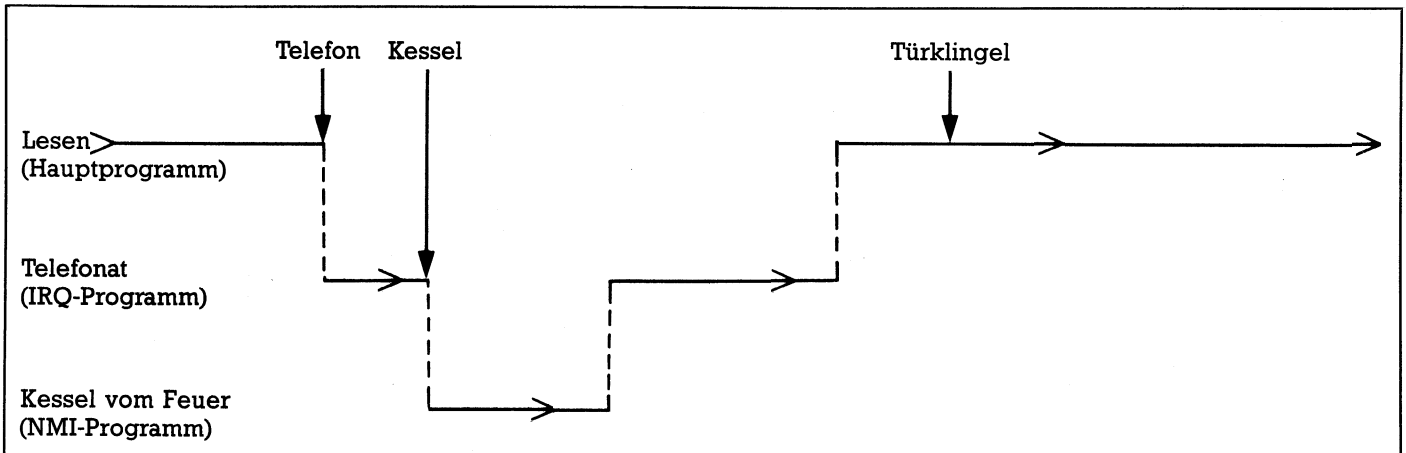


Bild 36. Unterbrechungsbeispiele im täglichen Leben

Artikels fort, fest entschlossen, sich nun nicht mehr unterbrechen zu lassen. Kurze Zeit später klingelt jemand an der Tür. Sie lassen sich dadurch nicht stören.

Dieses Gleichnis gibt ziemlich genau wieder, was sich im Computer – nur bei millionenfacher Geschwindigkeit – bei Unterbrechungen abspielt. In Bild 36 ist das Schema des Ablaufes grafisch dargestellt. In gewisser Weise ähnelt das ganze dem Abarbeiten von Unterprogrammsequenzen. Weshalb programmiert man dann nicht einfach mittels einiger JSR-Aufrufe? Dafür hat L.A.Leventhal einen einleuchtenden Vergleich: »Ein Unterbrechungs-System entspricht etwa einer Telefonklingel. Sie läutet, wenn ein Anruf empfangen wird, so daß man den Hörer nicht laufend abnehmen muß, um festzustellen, ob sich jemand in der Leitung befindet.« (L.A.Leventhal, »6502 Programmieren in Assembler«, München te-wi Verlag, Seite 12-1). Unterbrechungen können dann angefordert und abgearbeitet werden, wenn sie nötig sind, im Gegensatz zu Unterprogrammen, die erst dann berücksichtigt werden, wenn der Programmzähler einen JSR-Befehl erfaßt. Um also schnell reagieren zu können, müßte man sehr oft in einem Programm eine Unterroutine anspringen, die auf gewisse Registerinhalte prüft und dann zur Bearbeitung verzweigt oder – bei Nichtvorliegen einer Bedingung – im normalen Programm weiterfährt. Das kostet unnötig Zeit und Speicherraum. Mancher Verkehr des Computers mit Peripherie erfordert so schnelle Reaktionen, daß diese nur geleistet werden können durch Unterbrechen des laufenden Programmes.

Ich denke, daß Sie nun die Notwendigkeit von Unterbrechungen erkennen. Fast jede CPU kennt solche Unterbrechungssysteme. Man kann sie charakterisieren durch die Beantwortung folgender Fragen:

1) Welche Unterbrechungs-Eingänge weist die CPU auf?
2) Wie reagiert die CPU auf eine Unterbrechung?
3) Wie bestimmt die CPU die Unterbrechungsquelle, wenn die Anzahl der Quellen größer ist als die Anzahl der Eingänge?
4) Kann die CPU zwischen wichtigen und weniger wichtigen Unterbrechungen unterscheiden?
5) Wie und wann wird das Unterbrechungssystem freigegeben oder gesperrt?

All diese Fragen werden wir für unseren Computer ergründen.

48. Das Unterbrechungssystem der CPU 6510/6502

Einige dieser Charakteristika sind schnell zu zeigen:

Zu 1: Unsere CPU hat genau 2 Eingänge für Unterbrechungen (wenn man RESET außer acht läßt, was wir im folgenden meist tun werden).

Zu 3: Natürlich gibt es weitaus mehr denkbare Unterbrechungsquellen als diese 2 Eingänge, weshalb softwaremäßig eine Registerabfrage (das sogenannte Polling) durchgeführt wird, um die Quelle festzustellen.

Zu 4: Zwischen wichtiger und nicht so wichtiger Unterbrechung kann unsere CPU unterscheiden durch die Priorität der beiden Eingänge. Wir haben eine sogenannte maskierbare Unterbrechung, genannt IRQ, welche per Befehl ignoriert (maskiert) werden kann und eine andere, nicht maskierbare, die daher auch NMI (not maskable interrupt = nicht maskierbare Unterbrechung) genannt wird. NMI hat eine höhere Priorität als IRQ und kann deshalb für die wichtigeren Aufgabenstellungen eingesetzt werden.

Zu 5: Freigegeben oder gesperrt werden kann die IRQ-Unterbrechung durch ein Sperrbit (auch Maskenbit genannt), welches sich als Bit 2 im Flaggen-Register des Prozessors befindet. Das ist die I-Flagge. Für den Empfang der NMI-Unterbrechung kann die CPU nicht gesperrt werden.

Um mal die Parallele zu unserem Beispiel zu zeigen: Das Lesen des Artikels ist die gerade stattfindende Tätigkeit des Computers. Die Telefonklingel signalisiert einen IRQ, der im folgenden bearbeitet wird. Das Pfeifen des Teekessels soll einem NMI entsprechen. Wenn dieser dann bearbeitet ist, geht es mit der Abarbeitung des IRQ weiter. Nach Beendigung des Telefonates wird das Unterbrechungs-Sperrbit gesetzt (sie nehmen sich vor, sich nicht mehr stören zu lassen) und mit der normalen Tätigkeit fortgefahren. Weil der nun folgende IRQ damit maskiert ist, wird das Türklingeln ignoriert.

Die Frage 2, nämlich wie unsere CPU auf eine Unterbrechung reagiert, blieb noch unbeantwortet. Nun soll sie behandelt werden:

a) Am Ende jedes Befehls überprüft die CPU automatisch den Zustand des Unterbrechungs-Systems. Wenn an einer der beiden Unterbrechungsleitungen eine Anforderung vorliegt und diese auch freigegeben ist, beginnt die Unterbrechung zu wirken.
b) Zunächst wird der Programmzählerinhalt in der Reihenfolge MSB, LSB auf den Stapel geschrieben. Danach wandert noch der Prozessorstatus auf den Stapel (siehe Bild 37).
c) Durch Setzen des Unterbrechungs-Sperrbits I werden weitere maskierbare Unterbrechungen (IRQ) unterbunden.
d) Nun holt sich die CPU aus einem Vektor ganz am Ende des Speichers eine Adresse, lädt diese in den Programmzähler und startet auf diese Weise ein Serviceprogramm, das dem auslösenden Anlaß Rechnung trägt. In der Tabelle 19 sind die zu den Unterbrechungsformen und zum RESET gehörigen Vektoren aufgeführt.

Bevor wir uns weiter mit den so angesteuerten Routinen befassen, wollen wir die 4 Befehle kennenlernen, die uns noch fehlen.

49. Schlüssel zur Unterbrechungsprogrammierung: CLI, SEI, RTI, BRK

Das Sperren der maskierbaren Unterbrechung IRQ und das Löschen der Maske erfolgt durch Setzen oder Löschen des Sperrbits im Prozessorstatus-Register. Dieses Bit, die I-Flagge, kann durch den Befehl CLI gelöscht werden. **CLI**

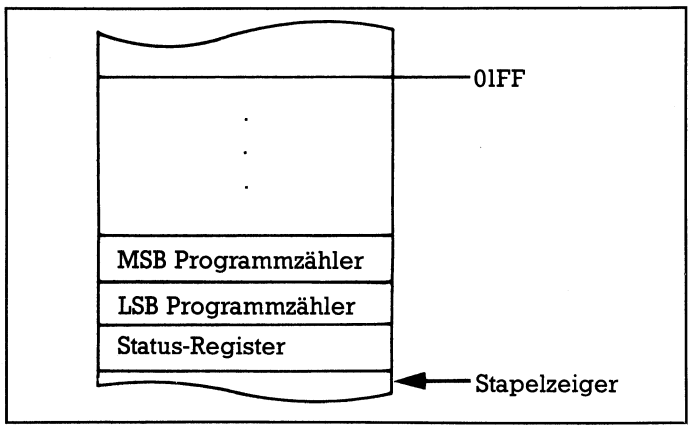


Bild 37. Die CPU rettet den Programmzähler und das Statusregister beim Eintreten einer Unterbrechung auf den Stapel

Unterbrechungsart	Vektor	Zieladresse
Maskierbare Unterbrechung (IRQ, BRK)	$FFFE/FFFF	65352 $FF48
Reset	$FFFC/FFFD	64738 $FCE2
Nichtmaskierbare Unterbrechung (NMI)	$FFFA/FFFB	65091 $FE43

Tabelle 19. Unterbrechungsvektoren und ihre Inhalte

kommt von »**CL**ear **I**nterrupt mask«, was bedeutet »lösche die Unterbrechungs-Maske«. Immer dann, wenn IRQs zugelassen sein sollen zur Bearbeitung durch den Mikroprozessor, muß damit die I-Flagge gelöscht werden. Wie Sie sehen, ist CLI ein 1-Byte-Befehl mit impliziter Adressierung. Er braucht genau 2 Taktzyklen zur Erledigung seiner Aufgabe.

Wenn wir später eigene Unterbrechungsroutinen schreiben, stehen wir oft vor der Frage, ob wir innerhalb unseres Unterbrechungsprogramms weitere Unterbrechungen zulassen wollen. Manchmal ist das wichtig, beispielsweise bei der Tastaturabfrage. Wie wir vorhin erwähnt haben, sperrt die CPU automatisch bei der Annahme von Unterbrechungen weitere IRQs durch Setzen der I-Flagge. Einer der ersten Befehle der eigenen Unterbrechungsroutine wird dann die Freigabe von Unterbrechungen sein durch Löschen der I-Flagge.

SEI bewirkt das Gegenteil von CLI. Der Befehl setzt die I-Flagge auf 1 (»**SE**t **I**nterrupt mask«) und verhindert, daß der Mikroprozessor weiteren IRQs seine Aufmerksamkeit schenkt. Das ist in den Fällen wichtig, in denen beispielsweise störungsfrei der Inhalt des Charakter-ROM gelesen werden soll oder während der Änderung von Speicherstellen, die die IRQ-Routine benutzt. Wie wichtig das Sperren von IRQs sein kann, haben Sie eventuell bemerkt, wenn Ihnen das Hilfsbildschirmprogramm aus Kap. 32 mal abgestürzt ist. Seit der letzten Folge – wo wir die IRQs gesperrt haben – ist Ihnen das sicherlich nicht mehr passiert. Ebenso wie CLI ist SEI ein 1-Byte-Befehl mit impliziter Adressierung, und auch er braucht 2 Taktzyklen zur Bearbeitung.

Noch eine Bemerkung zum Verhindern der IRQs. Wir werden später sehen, was alles während der 60mal pro Sekunde aufgerufenen Unterbrechung erledigt wird. Jede Routine, die SEI verwendet, verbraucht Rechenzeit. Wenn sie so lange dauert, daß eine oder mehrere dieser regelmäßigen IRQs unterbunden werden, kann das unter Umständen zu Störungen von Programmabläufen führen. In solchen Fällen ist es sinnvoll, in die eigene Routine den Teil der regulären IRQ-Routine einzubauen, der im Programmablauf durch sein Fehlen Störungen verursacht. Meistens kann man aber durch gute Planung eines Programmes dieses Problem umgehen.

RTI heißt »**R**e**T**urn from **I**nterrupt«, zu deutsch also: »kehre aus dem Unterbrechungsprogramm zurück.« Es entspricht in seinem Einsatz etwa dem RTS bei Unterprogrammrücksprüngen. Während RTS aber lediglich den alten Programmzählerinhalt vom Stapel holt (und noch eine 1 dazuaddiert), schafft RTI auch noch den alten Inhalt des Status-Registers vom Stapel zurück. Der genaue Ablauf ist wie folgt:

1) Alten Prozessorstatus vom Stapel wieder ins Status-Register schieben.
2) Stapelzeiger um 1 erhöhen
3) LSB des alten Programmzählers vom Stapel nehmen und zurückschreiben.
4) Stapelzeiger um 1 erhöhen
5) MSB des alten Programmzählers vom Stapel nehmen und zurückschreiben.
6) Stapelzeiger um 1 erhöhen.

Damit ist der Zustand vor der Unterbrechung wiederhergestellt. Auch die I-Flagge ist so automatisch wieder gelöscht, denn vor der Unterbrechung war sie sicher nicht gesetzt gewesen und der alte Status-Zustand ist ja jetzt wieder vorhanden.

RTI ist ebenfalls ein 1-Byte-Befehl mit impliziter Adressierung. Seine vollständige Bearbeitung dauert 6 Taktzyklen.

Bei eigenen Unterbrechungs-Routinen verwendet man häufig nicht RTI, sondern springt durch JMP an eine sinnvolle Stelle des normalen Unterbrechungsprogrammes. Auf diese Weise kann man dann die normalen Arbeitsgänge der vorprogrammierten Unterbrechung oder Teile davon noch ausführen lassen.

Den Befehl **BRK** (break=Software-Unterbrechung) haben wir schon verwendet. Er entspricht in seinem Einsatz etwa dem STOP-Befehl in Basic und dient wie jener Befehl dort hauptsächlich dem Testen von Programmen. Tatsächlich unterscheidet sich die Reaktion unserer CPU bei Auftreten eines BRK kaum von der bei einem IRQ. Folgendes passiert:

a) Der Programmzähler wird um 2 erhöht.
b) Bit 4 des Prozessorstatusregisters, die Break-Flagge B, wird auf 1 gesetzt.
c) Das MSB des Programmzählers wird auf den Stapel gebracht und der Stapelzähler um 1 heruntergezählt.
d) Dasselbe geschieht nun mit dem LSB des Programmzählers
e) und mit dem Statusregister.
f) Das Unterbrechungsmaskenbit, die I-Flagge, wird auf 1 gesetzt um IRQs zu sperren.
g) In den Programmzähler wird nun aus dem Vektor FFFE/FFFF dieselbe Adresse geladen, die auch bei IRQs benutzt wird. Damit startet nun das Programm, das diese Unterbrechung bearbeitet.

Sie sehen, daß der BRK-Befehl ein ziemlich komplizierter Geselle ist. Zwar handelt es sich wieder um einen 1-Byte-Befehl mit impliziter Adressierung, aber er benötigt immerhin 7 Taktzyklen, um all diese Arbeit zu bewältigen.

Wir haben BRK bisher immer zur Programmunterbrechung mit nachfolgender Registeranzeige durch den SMON eingesetzt. Der SMON ist – wie fast jeder Monitor – so programmiert, daß ein BRK zur Registeranzeige führt. Das ist natürlich sinnvoll beim Einsatz von BRK zur Fehlersuche. In dem Moment, wo ein BRK vom Prozessor bearbeitet wurde, kann nur durch die gesetzte B-Flagge von einem IRQ unterschieden werden. Es ist manchmal nötig, schon zu diesem Zeitpunkt diesen Unterschied festzustellen. Deshalb verwendet man den nachfolgend beschriebenen Test zu diesem Zweck:

```
    PLA
```

in den Akku wird das zuletzt auf den Stapel geschobene Prozessorstatus-Register geholt.

```
    PHA
```

und sogleich wieder zurückgeschoben

```
    AND #$10
```

durch die AND-Verknüpfung mit der Binärzahl 0001 0000 kann eine eventuell vorhandene B-Flagge isoliert werden.

```
    BNE BREAK
```

Falls eine B-Flagge gesetzt war, ist der Akku ungleich 0 und die Bearbeitung verzweigt zum von uns konstruierten BREAK-Programm. War der Akku nach dieser AND-Verknüpfung gleich 0, dann erfolgt keine Verzweigung und es handelt sich um einen IRQ, zu dessen Bearbeitung nun zu springen ist.

Es gibt noch eine andere – gebräuchlichere – Möglichkeit, zwischen einem BRK und einem IRQ zu unterscheiden, die allerdings erst zu einem späteren Zeitpunkt des computerinternen Unterprogrammes erfolgt. Von dieser zweiten Möglichkeit wird im SMON Gebrauch gemacht und wir werden sie nachher auch kennenlernen.

Natürlich kann der BRK-Befehl auch zu anderen Zwecken als zur Registeranzeige durch einen Monitor verwendet werden. Es kommt immer darauf an, welches Service-Programm wir dem Computer anbieten. Springt man aus so einem Service-Programm mittels RTI zurück ins Hauptprogramm, dann muß man berücksichtigen, daß der Programmzähler vor der Sicherung auf dem Stapel um 2 erhöht worden ist. Manchmal sind deshalb noch Korrekturen des Programms nötig.

Ich hoffe, daß Sie bisher diesen Artikel nicht zu frustrierend fanden, denn ständig ist die Rede vom eigenen Unterbrechungs-Programm und dabei wissen Sie – außer durch BRK – noch gar keine Möglichkeit, einen IRQ oder NMI auszulösen, und Sie sind sicher noch sehr vorsichtig mit dem Gedanken an eigene Unterbrechungs-Routinen, weil Ihnen ja noch

unbekannt ist, wie die normale Firmware Unterbrechungen behandelt. Keine Angst: All das werden wir noch klären. Betrachten Sie diesen Teil zum Thema Unterbrechungen vielleicht mehr wie ein Handbuch, in dem Sie dann, wenn Ihr Verständnis gestiegen ist, nochmal zurückblättern können.

Wir haben bisher nur betrachtet, wie unsere CPU reagiert, wenn an einem der beiden Unterbrechungs-Eingänge (IRQ und NMI) eine Unterbrechungs-Anforderung vorliegt. Um nun aber selbst ins Geschehen eingreifen zu können, ist es nötig zu wissen, wie diese Anforderung dorthin gelangt. Das erfordert von uns die Beschäftigung mit anderen Computerbausteinen als der CPU, die bisher im Mittelpunkt unseres Interesses stand.

50. Woher kommen die Unterbrechungs-Anforderungen?

Quellen für Unterbrechungen können viele genannt werden: Diskettenstation, Datasette, Drucker, Modem, Schaltelemente und so weiter. Um aber eine gewisse Übersicht zu bekommen, sollte man unterscheiden zwischen primären und sekundären Unterbrechungsquellen. Das soll kurz erläutert werden: Die Diskettenstation beispielsweise ist über den seriellen Port mit dem Computer verbunden. Dieser wiederum steht in direktem Kontakt zu 2 Bausteinen, den CIAs. Erst diese CIAs stehen in direktem Kontakt zur CPU. Alle Unterbrechungs-Quellen, die direkt Signale an die beiden Unterbrechungseingänge unserer CPU senden, sollen künftig »primäre« Quellen genannt werden, die anderen, die nur über solch eine primäre Quelle Unterbrechungs-Anforderungen stellen, werden von uns als »sekundäre« Quellen bezeichnet. Weil wir irgendwo einen Schnitt machen müssen – einmal, um nicht völlig auszuufern in der Erklärung von peripheren Geräten (das soll anderen, kompetenteren überlassen bleiben) und zum anderen, weil ich mich da auch nicht so gut auskenne – werden wir uns im folgenden auf die primären Unterbrechungsquellen beschränken. Da bleibt aber noch mehr als genug zu tun übrig und deshalb soll auch nur eine Auswahl dieser Primärquellen detailliert behandelt werden.

Welches sind nun die primären Unterbrechungsquellen? Hier sind sie aufgeführt:

1) Der VIC-II-Chip (MOS 6566/6567 Video Interface Controller)
2) Die beiden CIAs (MOS 6526 Complex Interface Adapter)
3) Die RESTORE-Taste
4) Der Expansion-Port
5) RESET (paßt hier nicht ganz her, woanders aber auch nicht besser)

Den Expansion-Port werden wir nicht behandeln und einen RESET nur ziemlich kurz betrachten, weil es sich dabei eigentlich nicht um eine Unterbrechung im bisher definierten Sinn handelt.

51. Der VIC-II-Chip als Unterbrechungsquelle

Soweit ich feststellen konnte, kommt der VIC-II-Chip in Bezug auf unsere CPU nur als Anforderer von maskierten Unterbrechungen (IRQ) in Frage. Die Handhabung seiner Unterbrechungs-Verlangen geschieht im VIC-II-Chip durch zwei Register. Vier Ereignisse sind eingeplant, deren Eintreten zur Unterbrechung führen kann:

1) Rasterzeilen-Unterbrechung
2) Kollision eines Sprites mit Hintergrund
3) Kollision von Sprites untereinander
4) Lichtgriffel-Unterbrechung.

Die ersten 3 Auslöser werden wir uns in kommenden Folgen genau ansehen und dabei vielerlei interessante Möglichkeiten feststellen. Die Option, die der Lichtgriffel bietet, wird nicht behandelt werden: Meine Kenntnisse auf diesem Sektor sind nur gering (nobody is perfect).

Das sogenannte Interrupt Enable Register (Unterbrechungs-Zulassungs-Register) des VIC-II-Chips ist Register 26. Es befindet sich in der Speicherstelle 53274 ($D01A) (siehe Bild 38).

In diesem Register wird festgelegt, ob eines – oder mehrere – der 4 möglichen auslösenden Ereignisse eine Unterbrechungsanforderung an den Mikroprozessor senden soll. Jedem Ereignis ist ein Bit zugeordnet. Ist dieses Bit gleich 1, dann ist die Unterbrechung freigegeben, ist es gleich 0, dann liegt eine Sperrung vor. Die Zuordnung der Bits ist wie folgt:

Bit 0 Rasterzeilen-IRQ
Bit 1 Sprite/Hintergrund-Kollision
Bit 2 Sprite/Sprite-Kollision
Bit 3 Lichtgriffel-IRQ
Bits 4 bis 7 sind ungenutzt und haben immer den Wert 1.

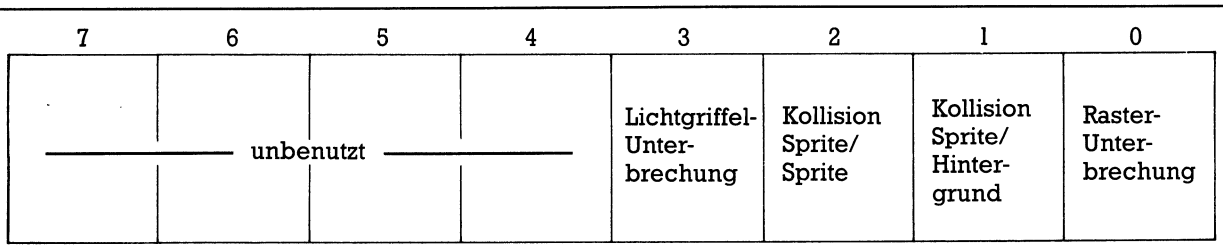


Bild 38. Das Interrupt-Enable-Register (53274 = $d01h) des VIC-II-Chip

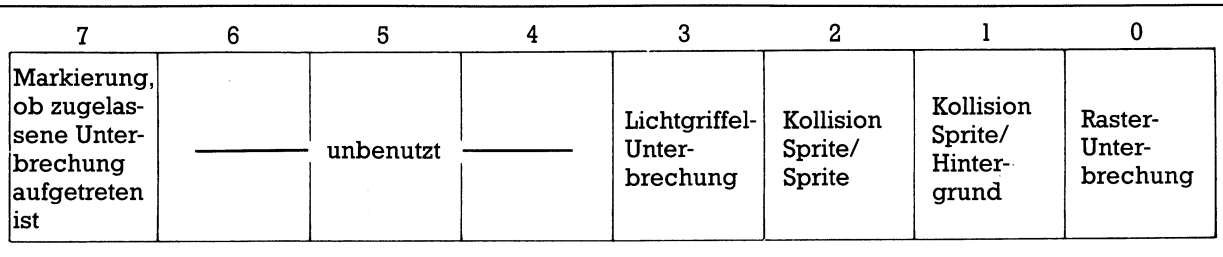


Bild 39. Das Interrupt-Latch-Register (53273 = $d019) des VIC-II-Chip

Das Register 25 wird Interrupt Latch Register genannt, was etwa zu übersetzen wäre mit »Unterbrechungs-Einrast-Register« (siehe Bild 39). Der englische Ausdruck »latch«, der nur umschreibend oder sehr technisch übersetzt werden kann, beschreibt eigentlich recht genau, was in diesem Register geschieht. Ein »latch« ist nämlich so etwas wie ein Schnappriegel, also ein Riegel, der bei der Betätigung einrastet. Wenn eines der 4 möglichen Ereignisse eintritt, schnappt im dazugehörigen Bit dieses Registers der Inhalt auf 1. Die Bit-Zuordnung ist die gleiche wie in Register 26. Aber das Bit 7 hat hier noch eine Bedeutung: Ist eines der Bits 0 bis 3 auf 1 gesetzt und das dazugehörige Ereignis in Register 26 auch zur Unterbrechung zugelassen (also auch dort gleich 1), dann taucht in Register 25, Bit 7 eine 1 auf. So kann durch einfaches Lesen dieses Bits festgestellt werden, ob ein IRQ durch den VIC-II-Chip ausgelöst wurde.

Will man in diesem Register ein gesetztes Bit löschen, muß man – außergewöhnlich! – eine 1 in die Bitposition schreiben.

Mit Recht erwarten Sie nun eigentlich eine Anwendung des bisher Gelernten. Bei Unterbrechungsprogrammen ist es aber dringend nötig, immer den gesamten Komplex im Auge zu haben. Ich habe mich daher entschlossen, zuerst alles zu erklären und dann Anwendungsmöglichkeiten vorzustellen. Ihre Geduld wird auf eine harte Probe gestellt, aber ich hoffe, daß Sie später feststellen, daß es sich gelohnt hat, etwas zu warten.

52. Die beiden CIA-Bausteine als Unterbrechungsquellen

An sich sind die beiden CIAs in unserem Computer völlig identisch. Sie werden aber unterschiedlich eingesetzt. Sehen wir uns zunächst einmal an, was beiden in Bezug auf Unterbrechungen gemeinsam ist, um danach die Unterschiede festzuhalten. Die Unterbrechungs-Steuerung geschieht in Register 13 dieser Bausteine. Dieses Register hat 2 Funktionen: Es bestimmt, ob eine Unterbrechungsanforderung an die CPU gesandt werden soll, und es stellt fest, ob ein Ereignis stattgefunden hat, das zur Unterbrechung führen kann. Die Bedienung dieses Registers ist demzufolge auch etwas unübersichtlich, aber wir haben schon ganz andere Probleme gemeistert.

Sehen wir uns zuerst einmal an, welche Ereignisse vom Standpunkt eines CIA-Bausteines als Unterbrechungskriterium dienen können:

1) Unterlauf der Uhr A
2) Unterlauf der Uhr B
3) Die interne Uhr hat eine Alarmzeit erreicht
4) Am SP-Eingang (hängt mit dem seriellen Port zusammen) ist ein bestimmter Zustand erreicht
5) An einem Eingang namens FLAG ist ein bestimmter Zustand erreicht.

Die Ereignisse 4 und 5 werden wir ebenfalls im weiteren weitgehend ausklammern.

Nun zum Register 13, dem Unterbrechungs-Kontroll-Register (siehe Bild 40).

Auch hier gehört zu jedem Ereignis ein Bit. Dabei – um Wiederholungen zu vermeiden – ist die Zuordnung schon durch die eben angegebene Ereignisaufzählung gegeben. Ziehen Sie von der vorangestellten Nummer immer eine 1 ab und Sie haben die Bitnummer. Die Bits 5 und 6 sind unbenutzt. Bit 7 hat eine dreifache Funktion, die eng mit den anderen Bitinhalten verknüpft ist. Sehen wir uns das mal der Reihe nach an:

Lesen des Registers

Sind Unterbrechungsereignisse aufgetreten, dann sind die dazugehörigen Bits auf 1 gesetzt. Bit 7 ist gleich 1, wenn mindestens ein solches Ereignis stattgefunden hat und außerdem dieses Ereignis als Unterbrechungsauslöser freigegeben ist. Auf diese Weise kann – ähnlich wie beim VIC-II-Chip-Register 25 – festgestellt werden, ob die Unterbrechung durch einen der beiden CIAs angefordert wurde. Im Unterschied aber zum VIC-II-Register wird Register 13 durch das Lesen gelöscht. Braucht man den Inhalt also noch, sollte man ihn irgendwo zwischenspeichern.

Schreiben in das Register

Bit 7 = 0 erzeugt Sperren.

Das erkennt man am besten an einem Beispiel. Nehmen wir an, wir möchten die Unterbrechung sperren, die durch einen Unterlauf von Uhr A erzeugt werden kann. Das betrifft das Bit 0. Wir schreiben in das Register 13 folgende Zahl: 0000 0001

Wie Sie sehen, ist das Bit 7 gleich 0. Die 1 in Bit 1 bewirkt die Sperrung. Durch die Nullen in den anderen Bits wird bewirkt, daß die anderen Unterbrechungs-Ereignisse nicht beeinflußt werden. Wollten wir alle sperren, dann müßten wir einschreiben: 0001 1111

Auf diese Weise können selektiv einzelne Unterbrechungen durch Einschreiben der 1 bei gelöschtem Bit 7 gesperrt werden.

Bit 7 = 1 erzeugt Freigabe.

Auch hier wieder ein Beispiel. Wenn wir ganz gezielt Unterbrechungen durch Unterlauf der Uhr A freigeben wollen, müssen wir die folgende Zahl in Register 13 schreiben: 1000 0001

Bit 7 (gleich 1) zeigt an, daß diejenigen Unterbrechungen freizugeben sind, deren Bits auf 1 gesetzt sind. Alle anderen Unterbrechungen, wo also in der dazugehörigen Bitposition der einzuschreibenden Zahl eine 0 steht, bleiben unverändert.

Ein wichtiger Unterschied zwischen den beiden CIAs ist der, daß der Unterbrechungsausgang von CIA 1 mit dem IRQ-Eingang der CPU verbunden ist, wohingegen der entsprechende Ausgang von CIA2 an den NMI-Eingang unseres Mikroprozessors führt. Daher löst der CIA 1 nur IRQs aus, er wird manchmal deshalb auch IRQ-CIA genannt. Der andere ist dann der NMI-CIA, weil er nur NMIs anfordern kann.

53. Der IRQ-CIA

Das Register 13 des IRQ-CIA (der die Speicherstellen 56320 bis 56335 belegt), liegt in Zelle 56333 ($DC0D). Die einzel-

7	6	5	4	3	2	1	0
Mehrfunktions-Bit	— unbenutzt —		FLAG-Eingang	SP-Eingang	ALARM bei interner Uhr	Unterlauf Uhr B	Unterlauf Uhr A

Bild 40. Genereller Aufbau der Unterbrechungs-Kontroll-Register (13) der beiden CIA-Bausteine

nen Bits sind wie folgt zugeordnet:
Bit 0 Unterlauf Uhr A
Von hier kommt der IRQ, der 60mal pro Sekunde stattfindet zur Tastaturabfrage, zum Weiterstellen der TI$-Uhr etc.
Bit 1 Unterlauf Uhr B
Spielt bei Kassettenoperationen und dem seriellen Port eine Rolle.
Bit 2 ALARM bei interner Uhr.
Spielt beim Zufallszahlengenerator (RND(0)) eine Rolle.
Bit 3 Hier kommen durch den User-Port Unterbrechungs-Anforderungen.
Bit 4 ist verbunden mit dem seriellen Port und der Kassetten-Lese-Leitung.

54. Der NMI-CIA

Ebenso kurz und schmerzlos wie beim CIA 1 soll auch das besondere am CIA 2, dem NMI-CIA (er belegt den Speicher von 56576 bis 56831) vorgestellt werden. Sein Register 13 findet sich in Speicherstelle 56589 ($DD0D). Die Bits 0 und 1 (Unterläufe der beiden Uhren) spielen beim Senden beziehungsweise Empfangen von Daten über die RS232C-Schnittstelle eine Rolle, Bit 2 (ALARM) wird nicht verwendet, Bit 3 ist direkt mit dem User-Port verbunden ebenso wie Bit 4. Der NMI-CIA wird uns in seiner normalen Funktion nicht mehr beschäftigen.

55. Die RESTORE-Taste und ein kleines Testprogramm

Die RESTORE-Taste ist direkt mit dem NMI-Eingang unseres Mikroprozessors verbunden. Das ermöglicht es uns, durch einfaches Drücken dieser Taste jederzeit ins Geschehen einzugreifen, ohne uns um Details kümmern zu müssen, ob sich der Computer gerade im Direkt- oder im Programm-Modus befindet und so weiter. Denn NMI hat die höchste Priorität der Unterbrechungen.

Ein kleines Testprogramm soll Ihnen hier noch vorgestellt werden, das Sie vielleicht aber noch nicht ganz verstehen werden, weil wir erst in den nächsten Kapiteln die eingebauten Serviceprogramme kennenlernen werden. Schalten Sie also den SMON ein und geben Sie das Programm 5 ein (ab $6000):

```
6000  PHA       mit diesen Befehlen retten wir Akku und Register auf
6001  TXA       den Stapel.
6002  PHA
6003  TYA
6004  PHA

6005  LDA #$7F  0111 1111 ist das in binär.
6007  STA $DD0D Dadurch werden alle NMIs, die vom CIA 2 kommen
                könnten, gesperrt. Erinnern Sie sich: Bit 7 ist Null
                beim Schreiben, also Sperrfunktion.
600A  LDY $DD0D Lesen des Registers 13 löscht dieses und zeigt uns,
                ob die NMI-Anforderung von dort kam.
600D  BMI $601A falls NMI-Anforderung vom CIA 2 kam, wird verzweigt
600F  LDA $D020 ansonsten kommt der NMI von der RESTORE-Taste,
                und in den Akku wird die Rahmenfarbe eingeladen
6012  EOR #$0E  Ausgehend davon, daß als Rahmenfarbe 14 vorliegt,
                wird diese exklusiv geORDERt zu Null. Ist die Rah-
                menfarbe 0, dann wird sie wieder 14.
6014  STA $D020 Einschreiben des neuen Farbwertes
6017  JMP $FEBC Sprung in den Rest der normalen NMI-Routine
601A  JMP $FE72 Sprung in die normale NMI-Routine für den Fall, daß
                die Anforderung durch den NMI-CIA kam.
```

Programm 5. Ein kleines Testprogramm demonstriert die Wirkung einer Unterbrechung: Durch Drücken der RESTORE-Taste wird die Rahmenfarbe geändert.

Am besten speichern Sie nun das Programm ab und schalten dann mittels dem SMON-Kommando M 0318 die Anzeige der Bytes ab $0318 ein. Dort steht in den beiden ersten Speicherzellen 47 und FE. Mit dem Cursor fahren Sie in diese Zeile und ändern den Inhalt in 00 und 60, also unsere Programmstartadresse in der LSB/MSB-Form. Nach einem RETURN läuft nun jede NMI-Anforderung über unser Programm. Nun können Sie es ausprobieren, indem Sie mal die RESTORE-Taste drücken. Es genügt völlig, alleine diese Taste zu betätigen. Das wirkt – sichtbar durch die Änderung der Rahmenfarbe – in jedem Modus und jederzeit. Eine kleine Merkwürdigkeit ist, daß man manchmal etwas Geduld aufbringen muß, bis man die Wirkung sieht. Ich vermute, daß der NMI so schnell erledigt wird, daß sich mehrer NMIs pro Tastendruck ereignen. Man müßte sich noch eine kleine Routine überlegen, die die Wirkung etwas verzögert, denn 2 solche EOR-Kommandos nacheinander heben sich gegenseitig auf. Zum Schluß noch eine Aufstellung (Tabelle 20) mit den besprochenen Befehlen.

56. Der normale Verlauf eines IRQ

Neulich hatten wir bereits festgestellt, daß eine IRQ-Anforderung (nach dem Retten des Programmzählers und des Prozessorstatus-Registers, sowie dem Setzen der I-Flagge) den Inhalt des Vektors $FFFE/FFFF in den Programmzähler holt. Dort steht die Adresse $FF48(dez. 65352) und deshalb startet nun das dort im ROM verankerte Programm, welches wir uns nun im einzelnen ansehen werden (alle Adressen als Dezimalzahlen, in Bild 41 finden Sie das Flußdiagramm dazu)

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktcyclen	Beeinflussung von Flaggen
CLI	implizit	1	58	88	2	I-Flagge
SEI	implizit	1	78	120	2	I-Flagge
RTI	implizit	1	40	64	6	alle Flaggen
BRK	implizit	1	00	0	7	B-Flagge vor dem Schieben auf den Stapel, I-Flagge danach

Tabelle 20. Die Daten zu den letzten Assembler-Befehlen

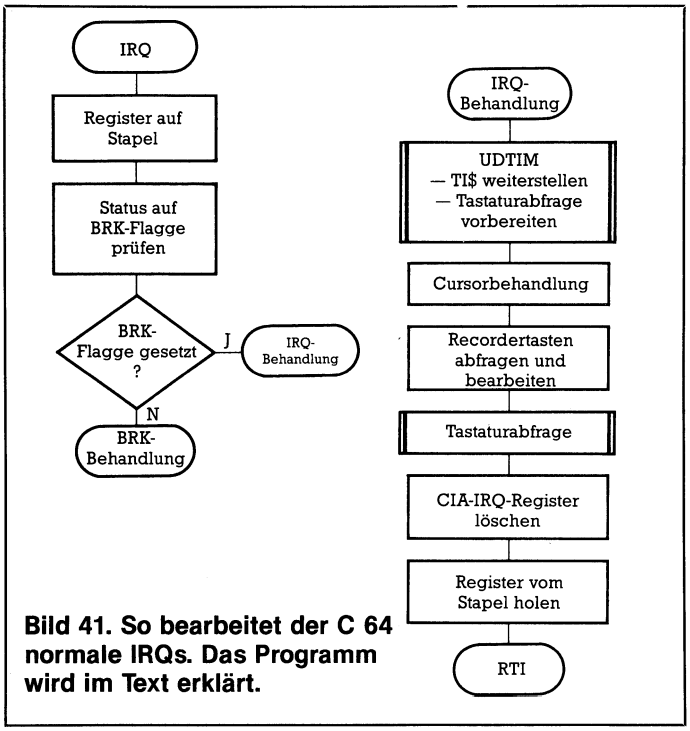


Bild 41. So bearbeitet der C 64 normale IRQs. Das Programm wird im Text erklärt.

```
65352   PHA        Zunächst werden der Akku und
        TXA        die Register X und Y auf den Sta-
        PHA        pel geschoben
        TYA
        PHA
```

Trickreich sind die beiden folgenden Befehle, mit denen das zu Beginn durch die CPU gerettete Statusregister gelesen wird:

```
        TSX        Stapelzeiger ins X- Register
        LDA 260,X  Einladen des Status-Registers
```

Nun wird geprüft, ob die BRK-Flagge gesetzt ist. Wenn das der Fall ist, dann ist der Auslöser ein BRK gewesen, ansonsten ein IRQ:

```
        AND #16    Isolieren der BRK-Flagge
        BEQ 65368  Wenn keine BRK-Flagge, dann
                   überspringen des nächsten
                   Befehls.
65365   JMP (790)  Falls BRK
65368   JMP (788)  Falls IRQ
```

Den vorletzten Sprungbefehl werden wir bei der BRK-Behandlung verfolgen. Interessant für uns ist jetzt der indirekte Sprung bei 65368. Der Vektor 788/789 ($314/315) liegt im RAM! Damit können wir ihn auf eigene Routinen verstellen. Genau hier ist der Ansatzpunkt für nahezu alle Eingriffe in die Unterbrechungsbehandlung. Der voreingestellte Wert in diesem Vektor ist die Adresse 59953 ($EA31). Das dort angesiedelte Programm wird im Normalfall 60 mal in der Sekunde ausgeführt:

```
59953   JSR 65514  Das ist ein Kernel-Sprungbefehl
                   zur Routine UDTIM bei 63131.
```

In diesem Unterprogramm wird zuerst die Uhr TI$ weitergestellt und dann die Tastaturabfrage vorbereitet.

```
59956              In diesem Programmteil erfolgt
bis                die Cursorbehandlung.
60000
60001              Anschließend wird abgefragt, ob
bis                eine Recordertaste gedrückt ist
60026              und entsprechende Flaggen
                   bearbeitet.
60027   JSR 60039  Dieses Unterprogramm dient zur
                   Tastaturabfrage.
```

Auch in dieser Routine tritt übrigens ein indirekter Sprung nach einem RAM-Vektor auf (655/656 = $28F/290), der normalerweise auf 60232 zeigt, aber auch auf eine eigene Routine verbogen werden könnte.

Enthalten in der Tastaturabfrage ist auch die Überprüfung der RUN/STOP-Taste, die aber nur zusammen mit den in dem UDTIM-Aufruf voreingestellten Flaggen funktioniert. Deshalb wird das Abschalten der RUN/STOP-Taste im allgemeinen dadurch durchgeführt, daß man den IRQ-Vektor auf 59956 stellt und damit den ersten JSR-Befehl überspringt. Allerdings wird auf diese Weise auch die TI$-Uhr nicht weitergestellt.

```
60030   LDA 56333  Das ist das Unterbrechungs-
                   Kontrollregister des IRQ-CIA, das
                   hier durch Auslesen gelöscht
                   wird.
```

Den Abschluß der IRQ-Routine bildet nun noch das Zurückschreiben der Register:

```
60033   PLA        Zurückholen des
        TAY        Y- und
        PLA
        TAX        des X-Registers
        PLA        sowie des Akku.
60038   RTI        Damit kehrt der Computer zu
                   dem durch den IRQ unterbroche-
                   nen Programm zurück.
```

Somit hätten wir's. Nun können wir je nach Bedarf entscheiden, welche von diesen Servicetätigkeiten wir bei einem eigenen IRQ-Programm brauchen: Die Uhr TI$, die Cursorbehandlung, die Abfrage der Recordertasten und die Tastaturabfrage.

Sehen wir uns nun an, was geschieht, wenn ein BRK-Kommando der Auslöser war.

57. BRK-Unterbrechung

Wir hatten vorhin am Scheideweg zwischen IRQ und BRK den letzteren links liegen gelassen. Normalerweise verwendet man beim Programmieren in Assembler ja ein Software-Instrument wie zum Beispiel den SMON, der so gebaut ist, daß der BRK-Vektor, welchen wir vorhin kennengelernt haben ($316/317 = 790/791) auf die Registeranzeige weist. Was geschieht eigentlich, wenn der BRK-Vektor unverändert bleibt, so also, wie er im Einschaltzustand des Computers vorliegt?

Dann zeigt er auf die Adresse 65126 ($FE66), wo ein Teil der NMI-Routine zu finden ist (Siehe auch das Flußdiagramm in Bild 42):

```
65126   JSR 64789  Sprung ins Programm RESTOR,
                   in dem alle Vektoren (788-819)
                   gemäß einer ROM-Liste auf ihre
                   Ausgangswerte gesetzt werden.
        JSR 64931  Sprung in das Programm
                   I/O-RESET.
```

In diesem Programm werden die beiden CIAs auf die Anfangswerte gestellt.

```
        JSR 58648  Sprung in ein Programm, welches
                   zuerst den VIC-II-Chip initialisiert,
                   dann einen Bildschirmeditor-
                   RESET durchführt.Nach Beenden
                   dieser Routine ist der Bildschirm
                   gelöscht.
        JMP (40962)
```

Mit diesem indirekten Sprung ist die BRK-Unterbrechung beendet. Man sieht aber jetzt schon deutlich, daß es sich hier nicht um eine Unterbrechung im eigentlichen Sinn handelt,

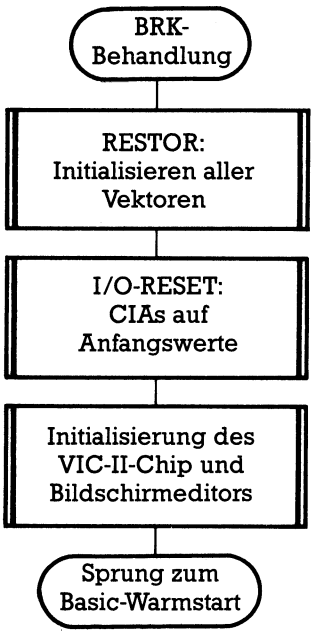


Bild 42. Auf diese Weise verläuft ein unvorhergesehener BRK im Sande.

vielmehr um einen Abbruch. In 40962/40963 steht die Adresse des Basic-Warmstarts (58235). Danach befindet sich der Computer im READY-Zustand in der Eingabe-Warteschleife.

Das Zurückholen der Register und ein RTI erübrigt sich hier, weil ohnehin viele Werte aus dem unterbrochenen Programm inzwischen weitgehend zerstört sind und alle Unterbrechungskontrollregister (CIAs und VIC-II-Chip) neu belegt wurden. Ein unkontrollierter BRK hat also recht fatale Folgen!

58. Was macht ein NMI?

Wenden wir uns nun der Firmware zu, die zur Bearbeitung eines NMI vorgesehen ist (Dazu sehen Sie sich bitte in Bild 43 das Flußdiagramm an).

In den letzten Kapiteln erfuhren wir, daß auch für diese Unterbrechung am Ende des Speichers ein Vektor vorhanden ist, nämlich $FFFA/FFFB (65530/65531). Dort steht die Adresse 65091 ($FE43), die nun in den Programmzähler gelangt und damit startet das folgende Programm:

65091 SEI Unterbrechungen niedrigerer Priorität werden gesperrt.

JMP (792)

Das ist nun wieder ein für uns sehr interessanter Vektor 792/793 ($318/319), der — weil er im RAM-Bereich liegt — verstellbar ist. Genau das haben wir am Ende der letzten Folge getan mittels des M-Kommandos von SMON um den NMI zu testen, den wir mit der RESTORE-Taste ausgelöst haben. Der vorher eingestellte Wert in diesem Vektor ist die Adresse 65095 ($FE47), also direkt der nächste Befehl nach dem indirekten Sprungbefehl.

65095 PHA Ebenso wie vorhin beim IRQ werden hier die Inhalte des Akku und der
TXA
PHA
TYA
PHA Register auf den Stapel geschoben.
LDA #127 das ist binär 01111111.
STA 56589 Sperrt alle weiteren NMI-Anforderungen
LDY 56589 NMI-CIA Kontrollregister laden.
BMI 65138 Wenn der NMI von der RESTORE-Taste kam, ist Bit 7 des Registers = 0, sonst = 1 (bei NMI-Anforderung durch NMI-CIA). Wenn also nicht durch die RESTORE-Taste, erfolgt Sprung.

An dieser Stelle läuft nun das Programm weiter, wenn die RESTORE-Taste der NMI-Auslöser war:

65110 JSR 64770 Das ist ein Unterprogramm, welches prüft, ob ein Modul ab $8000 vorhanden ist.

Dies wird dadurch angezeigt, daß von $8004 bis $8008 die Werte stehen: 195, 194, 205, 56, 48 (das ist »CBM80«).

BNE 65118 Wenn kein Modulprogramm ab $8000 vorliegt, erfolgt ein Sprung.
JMP (32770) Falls Modul.

Wenn ein Modul angezeigt wurde, erfolgt der indirekte Sprung nach dem Vektor $8002/8003, der vom Modul vorgegeben wird. Das kann man auch nutzen, um eigene Maschinenprogramme zu starten durch einen Druck auf die RESTORE-Taste. Man muß dann nur in die Speicherstellen $8002 bis 8008 die geforderte Zieladresse beziehungsweise »CBM80« schreiben.

Der nun folgende Abschnitt wird nur angesprungen, wenn die RESTORE-Taste der NMI-Auslöser war:

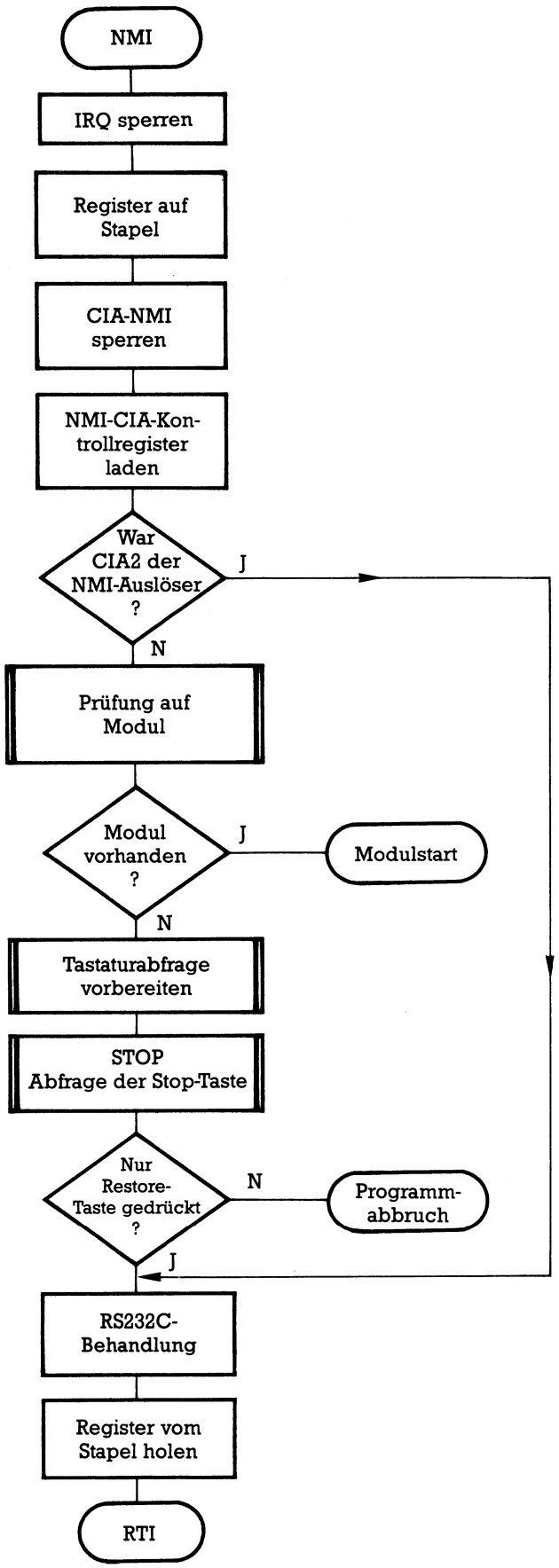


Bild 43. Flußdiagramm zum Ablauf einer NMI-Unterbrechung.

65118 **JSR 63164** Das ist ein Programmteil, der auch schon von der IRQ-Routine (nach dem Weiterstellen von TI$) durchlaufen wird. Hier werden einige Voreinstellungen für die Tastaturabfrage erledigt, die insbesondere die RUN/STOP-Taste betreffen.

JSR 65505 Kernelroutine STOP.

Dort befindet sich ein indirekter Sprung über den Vektor 808/809 ($328/329), also auch ein verstellbarer RAM-Vektor. Im Normalfall zeigt dieser Vektor auf 63213 ($F6ED). Dort wird geprüft, ob die RUN/STOP-Taste gedrückt ist. Eine andere Methode zum Ausschalten des RUN/STOP bietet sich hier an, die die Uhr TI$ ungeschoren läßt.

BNE 65138 Falls nur die RESTORE-Taste (also ohne RUN/STOP) gedrückt ist, erfolgt nun ein Sprung.

Waren aber sowohl die RUN/STOP- als auch die RESTORE-Taste gedrückt, dann folgt nun ein Programmabbruch, der uns schon von BRK her bekannt ist. Hier wie dort endet das Ganze dann mit dem Reset der I/O-Bausteine, des VIC-II-Chips, der Vektoren, des Bildschirmeditors und das Ergebnis ist ein Basic-Warmstart.

Ab 65138 befindet sich der Rest der NMI-Routine, auf die das Programm läuft, wenn

1) die NMI-Anforderung nicht von der RESTORE-Taste kommt oder

2) zwar von dieser Taste kommt, aber die RUN/STOP-Taste nicht gedrückt ist.

65138 bis 65211 Dieser ganze Abschnitt ist zur Behandlung der RS232C-Schnittstelle eingerichtet.

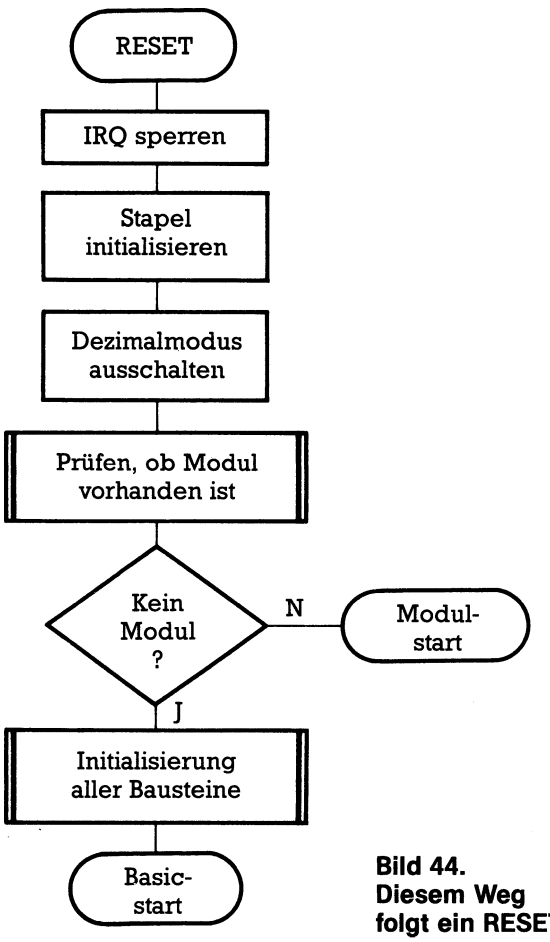


Bild 44. Diesem Weg folgt ein RESET

65212 **PLA** Abschluß des NMI durch Rückschreiben des Akku und der
TAY **PLA** Register vom Stapel
TAX
PLA

65217 **RTI** Rückkehr zum unterbrochenen Programm.

Wenn Sie sich nun mal unser kleines Demo-Programm aus Kapitel 55 ansehen, dann werden Sie feststellen, daß der Programmteil bis $600E lediglich den ersten Teil der normalen NMI-Routine kopiert. Die Prüfung auf das Modul und die RUN/STOP-Taste werden übersprungen. Statt dessen erfolgt nach der Abarbeitung des für die RESTORE-Taste gebauten Programmes das Ende der NMI-Routine ($FEBC = 65212). Im anderen Fall, wenn also die RESTORE-Taste nicht der Auslöser des NMI war, wird in die normale Routine ab 65138 eingemündet.

59. Eigentlich keine Unterbrechung: RESET

Weil wir alle Unterbrechungen hier bearbeiten wollen, soll auch der RESET angesprochen werden. Es handelt sich dabei aber nicht um eine Unterbrechung im bisher definierten Sinn. Mir fällt allerdings kein Platz ein, wo der RESET besser hinpassen würde. Ähnlich wie bei NMI und IRQ wird auch hier ein Vektorinhalt in den Programmzähler geladen, der in den höchsten Speicheradressen zu finden ist (Auch hierzu wieder ein Flußdiagramm in Bild 44).

Dieser Vektor liegt in $FFFC/FFFD. Der Inhalt ist die Adresse 64738 ($FCE2) und genau dort geht das Programm dann weiter:

64738 **LDX #255** Im ersten Teil wird der Stapelspeicher initialisiert.
SEI Verhindern von IRQ
TXS Stapelzeiger auf $FF
CLD Dezimal-Modus ausschalten (falls er eingeschaltet war).
JSR 64770 Das ist wieder das Unterprogramm, das auf ein Modul prüft.

Hier ergibt sich die Möglichkeit, auch beim RESET einzugreifen, indem man die Kennung CBM80 an die abgefragten Orte packt.

BNE 64751 Falls kein Modul, erfolgt Sprung.
64748 **JMP(32768)**

Dieser indirekte Sprung erfolgt nach dem Vektorinhalt von $8000/8001 = 32768/32769. Das ist ein anderer Vektor als wir ihn vorhin beim NMI hatten (dort war es $8002/8003 = 32770/32771). So kann ein anderer Programmteil angesteuert werden als durch den NMI, was übrigens auch dringend erforderlich ist, weil der Stapelzeiger zerstört wurde.

64751 Hier läuft das Programm weiter, falls keine Modulkennung erkannt wurde.

Der ganze Rest dient dem Versetzen des Computers in den Einschaltzustand. Allerdings bin ich davon überzeugt, daß noch irgendein Unterschied bestehen muß zwischen dem einfachen Aus- und wieder Anschalten des Computers und einem RESET. Es hat sich nämlich bei einigen Programmen gezeigt, daß sie nach einem RESET fehlerhafte Verläufe nehmen können, was nach einem totalen Aus- und wieder Anschalten nicht zu beobachten war. Der Grund für diesen Unterschied liegt (für mich) noch im Dunkeln. Vielleicht weiß das ja jemand von Ihnen. Dann schreiben Sie doch mal!

60. Die Sache mit dem Modulstart

Sowohl beim RESET als auch beim NMI haben wir festgestellt, daß der Modulstart-Bereich ab $8000 eine besondere Rolle spielt. In Bild 45 finden Sie nochmal zusammengefaßt, was sich dort findet wenn ein Modul vorhanden ist.

Wir wollen im folgenden Beispielprogramm (Programm 6) ein Modul simulieren, indem wir den SMON mittels des RESET anspringen. Der NMI — also die RUN/STOP-RESTORE-Tastenkombination — soll dabei wirkungslos gemacht werden.

Bild 46 zeigt ein Flußdiagramm dieses Beispielprogrammes:

Achten Sie bitte darauf, daß Sie nach dem Eintippen des Programmes abspeichern und — natürlich — daß die SMON-Version ab $C000 im Speicher vorliegt. Wenn Sie nun mal die RESTORE-Taste — oder RUN/STOP und RESTORE — drücken, passiert offensichtlich nichts. Das liegt daran, daß unser Programm lediglich die auf den Stapel gelegten Register wieder zurückholt und aus der Unterbrechung mit RTI ins normale Geschehen zurückkehrt.

Haben Sie einen RESET-Taster eingebaut? Dann drücken Sie doch mal drauf. Zunächst erkennen Sie den normalen RESET-Verlauf. Dann meldet sich aber nicht wie gewohnt die Nachricht CBM-Basic..., sondern der SMON mit einer Registeranzeige. Das RESET-Programm ab $602E folgt dem Firmware-Programm. Auf diese Weise (und mittels eines AUTOSTART) sichern sich Softwarehäuser manchmal gegen unbefugtes Kopieren ihrer Programme.

```
100  -           .LI 1,3
110  -           .BA $8000
112  -;
114  -;*******************************
116  -;  MODULSIMULATION
117  -;*******************************
118  -;
120  -           .EQ INITCZ=$E3BF
130  -           .EQ INITMS=$E422
140  -           .EQ INITV=$E453
150  -           .EQ SCREENCLR=$E544
160  -           .EQ RAMTEST=$FD50
170  -           .EQ IORESET=$FDA3
180  -           .EQ TVTAKT=$FF5B
190  -           .EQ RESTOR=$FF8A
192  -;
194  -; ****** MODULKENNUNG UND -VEKTOREN ****
196  -;
200  -           .WO RESET,NMI
210  -           .BY $C3,$C2,$CD,$38,$30
212  -;
214  -;******  RESET-PROGRAMM ********
216  -;
220  -RESET      STX $D016       ;RESET-BIT
230  -           JSR IORESET
240  -           JSR RAMTEST
250  -           JSR RESTOR
260  -           JSR TVTAKT
270  -           CLI
280  -           JSR INITV
290  -           JSR INITCZ
300  -           JSR INITMS
310  -           JSR SCREENCLR
320  -           JMP $C000       ;SPRUNG IN SMON
322  -;
324  -; ** NMI-PROGRAMM (RESTORE-TASTE) **
330  -NMI        PLA
340  -           TAY
350  -           PLA
360  -           TAX
370  -           PLA
380  -           RTI
390  -           .SY 1,4
```

Programm 6. Simulation eines Moduls

61. Nutzung der Unterbrechungen

Sowohl was die Hardware als auch die Firmware für die Unterbrechungsbehandlung angeht, haben wir nun einen guten Überblick gewonnen. Es ist jetzt an der Zeit, daß wir uns ansehen, auf welche Weise man dieses Reservoir an vielfältigen Möglichkeiten für sich nutzen kann. Dazu soll uns ein Überblick dienen:

I) Auslösung der Unterbrechung durch Hardware-Einwirkungen.

Da hätten wir beispielsweise den Userport oder den Expansion-Port, über die wir per CIAs Unterbrechungen anfordern können. Um es gleich zu sagen: Damit werden wir uns nicht auseinandersetzen. Meine Kenntnisse auf diesem Gebiet sind zu dünn. Aber vielleicht verstehen Sie das auch mal als Aufforderung, Ihre Versuche dazu anderen zu offenbaren? Also: Schreiben Sie doch mal!

II) Unterbrechungsauslösung per Software:

Damit haben wir immer noch ein weites Feld von Möglichkeiten vor uns:

IIa) Vorgesehene Nutzungen des IRQ
— mittels des VIC-II-Chips.

Speicherplatz ($)	8000	8001	8002	8003	8004	8005	8006	8007	8008
Inhalt	LSB	MSB	LSB	MSB	C	B	M	8	0
	RESET-Vektor		NMI-Vektor						

Bild 45. Diesen Inhalt müssen die Speicherstellen $8000 bis $8008 haben, damit ein Modulstart stattfindet.

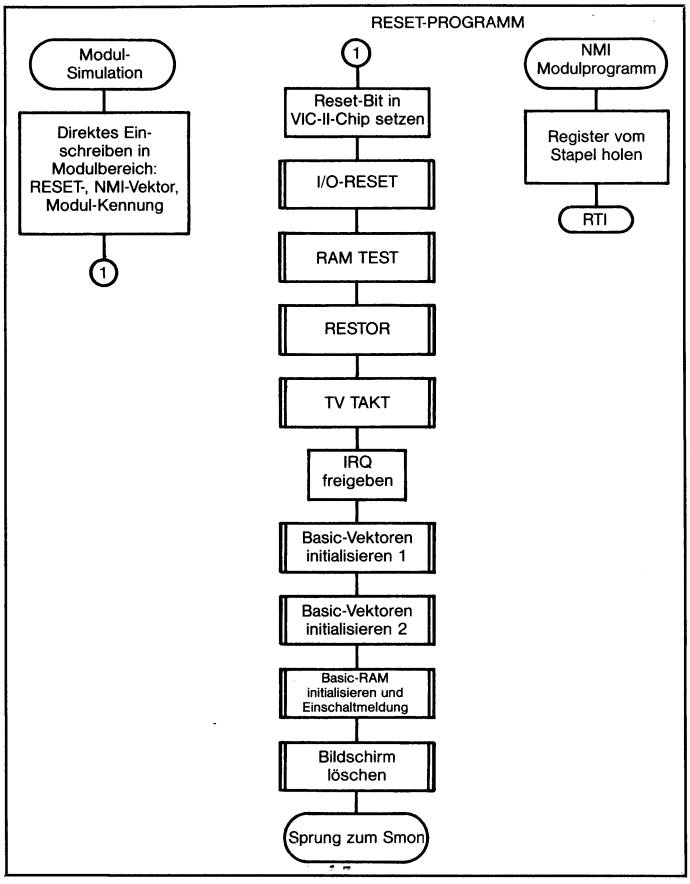


Bild 46.
Flußdiagramm zum Programm 6, Modulsimulation.

Da können wir uns auf den Rasterzeileninterrupt, die Sprite/Hintergrund- oder die Sprite/Sprite-Kollision stützen.
— oder mit Hilfe des CIA1

Da ist es vor allem der 60mal pro Sekunde auftretende Timer A-Unterlauf, der uns interessieren soll.

IIb) Vorgesehene Nutzungen des NMI

— CIA2: Läßt man die RS232C-Schnittstellenbehandlung außer acht, dann gibt es keine vorgesehene Nutzung.
— RESTORE: Zusammen mit der RUN/STOP-Taste kann man die vorgegebene Routine verändern, wie wir es schon in einigen Beispielen gezeigt haben.

Wir können außerdem noch unterscheiden zwischen Nutzungen, die periodisch stattfinden sollen (zum Beispiel eine spezielle Tastaturabfrage) und solchen, die stochastisch (= zufallsabhängig) oder willkürlich erfolgen (zum Beispiel Drücken der RESTORE-Taste). Beides ist auch durchführbar bei:

IIc) Nicht vorgesehene Nutzung der Unterbrechungen.

Da bietet sich vor allem der meistens völlig brach liegende CIA2 an mit seinen beiden Timern und der Alarmfunktion.

Wenn Sie aber erst einmal vertraut sind mit der Unterbrechungs-Programmierung und auch etwas Zeit zum Tüfteln investieren, finden Sie bestimmt noch eine ganze Menge weiterer Möglichkeiten.

Bei mehreren gleichartigen Unterbrechungsanforderungen (zum Beispiel IRQs) muß noch ein Weg gefunden werden, wie zwischen den dann vielleicht anfallenden unterschiedlichen Service-Routinen differenziert werden kann. Denkbar wären beispielsweise Aufgabenstellungen wie:

Jeder 3. Timer-IRQ soll den Joystick abfragen, oder RESTORE+h soll den Hilfsbildschirm zeigen, RESTORE+z soll den aktuellen Bildschirm wieder restaurieren, etc.

```
PROGRAMM 7

,6000  78        SEI
,6001  A9 28     LDA #28
,6003  8D 14 03  STA 0314
,6006  A9 60     LDA #60
,6008  8D 15 03  STA 0315
,600B  A9 F8     LDA #F8
,600D  8D 12 D0  STA D012
,6010  AD 11 D0  LDA D011
,6013  29 7F     AND #7F
,6015  8D 11 D0  STA D011
,6018  A9 81     LDA #81
,601A  8D 1A D0  STA D01A
,601D  A9 00     LDA #00
,601F  8D 20 D0  STA D020
,6022  A9 04     LDA #04
,6024  85 02     STA   02
,6026  58        CLI
,6027  60        RTS
------------------------
,6028  AD 19 D0  LDA D019
,602B  8D 19 D0  STA D019
,602E  30 07     BMI 6037
,6030  AD 0D DC  LDA DC0D
,6033  58        CLI
,6034  4C 31 EA  JMP EA31
------------------------
,6037  AD 12 D0  LDA D012
,603A  C9 F8     CMP #F8
,603C  B0 11     BCS 604F
,603E  18        CLC
,603F  65 02     ADC   02
,6041  8D 12 D0  STA D012
,6044  A0 03     LDY #03
,6046  88        DEY
,6047  D0 FD     BNE 6046
,6049  EE 20 D0  INC D020
,604C  4C 81 EA  JMP EA81
------------------------
,604F  A9 00     LDA #00
,6051  8D 20 D0  STA D020
,6054  A9 32     LDA #32
,6056  8D 12 D0  STA D012
,6059  4C 81 EA  JMP EA81
------------------------
,605C  78        SEI
,605D  A9 00     LDA #00
,605F  8D 1A D0  STA D01A
,6062  A9 31     LDA #31
,6064  8D 14 03  STA 0314
,6067  A9 EA     LDA #EA
,6069  8D 15 03  STA 0315
,606C  A9 0E     LDA #0E
,606E  8D 20 D0  STA D020
,6071  58        CLI
,6072  60        RTS
------------------------
```

Programm 7. Das im Artikel entwickelte Programm auf einen Blick

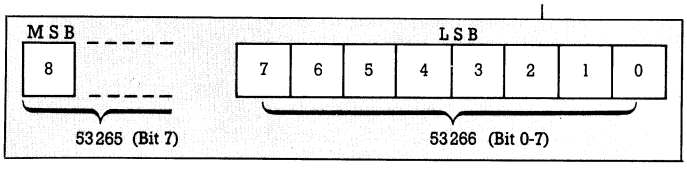


Bild 47. So sieht das 9-Bit-Register im VIC-II-Chip aus, welches die Rasterzeilen mitzählt.

Sie sehen, eine große Menge Arbeit wartet auf uns. Nicht zu allen Möglichkeiten werde ich hier Beispielprogramme zeigen. Außerdem dürfen die dann auch nicht zu undurchsichtig sein und man sollte möglichst den Erfolg eines solchen Demo-Programmes auf dem Bildschirm erkennen können. Trotzdem hoffe ich, daß die nachfolgend und in den nächsten Kapiteln gezeigten Programmlösungen ausreichen, Ihnen die Unterbrechungs-Behandlung mit eigenen Routinen durchschaubar zu machen. Ich will Ihnen aber nicht verschweigen, daß auch mir noch längst nicht alle Geheimnisse der Unterbrechungsprogrammierung offenbar geworden sind. Oft finde ich mich unversehens in Programm-Sackgassen wieder. Das soll Ihnen als kleiner Trost dienen, wenn Sie mal nach dem 1001. Absturz müde und mit rauchendem Kopf vor Ihrem Commodore-Ungeheuer sitzen.

62. Ein Programm zum VIC-II-IRQ

Sehr schöne Effekte lassen sich durch eine periodische IRQ-Anforderung per Rasterzeileninterrupt mittels des VIC-II-Chip erzielen. Deshalb ist sowas auch ein beliebtes Objekt für Demos von Unterbrechungsprogrammen. Als Ziel setzen wir uns, einen Bildschirm zu konstruieren, dessen Rahmen in allen Farben schillert.

Leser der Grafikserie werden diese Möglichkeit des VIC-II-Chip schon kennen: Man kann dem Kathodenstrahl, der über den Monitor huscht, um das Bild zu erzeugen, über zwei Register folgen, die Rasterregister, wo jede Rasterzeile mitgezählt wird. Ohne an dieser Stelle allzusehr auf die Einzelheiten einzugehen, soll hier nur bemerkt werden, daß die Numerierung dabei etwa von 0 bis 280 geht, weil auch der Rahmen und nicht sichtbare Teile des Bildschirmes vom Strahl überstrichen werden. Wo das Textfeld anfängt, ist von Monitor zu Monitor (oder Fernseher) etwas unterschiedlich. Bei mir beginnt es oben in Rasterzeile 50 und endet unten bei Zeile 248. Sollten die im Beispielprogramm 7 (Programm 7) nachher voreingestellten Randwerte bei Ihnen also anders sein, können Sie sie durch einige später noch angegebenen POKEs ändern. Die beiden Rasterzeilenregister sind:

$D012 (53266)
$D011 (53265)

Von $D011 allerdings ist nur das Bit 7 als msb der Rasterzeilenzahl für uns von Bedeutung. Bild 47 soll diese Belegung deutlich machen:

Das Interessante an diesen Registern ist nun, daß man auch in sie schreiben kann. Die auf diese Weise festgelegte Rasterzeile ist dann der Auslöser des IRQ, falls dieser im Interrupt-enable-Register $D01A freigegeben wurde (das kennen wir noch aus Kapitel 51).

Damit kann also unsere primäre Unterbrechungsquelle (der VIC-II-Chip) programmiert werden. Halten wir die zwei Schritte dazu nochmal fest:

1) Rasterzeile festlegen, bei der ein IRQ ausgelöst werden soll, durch Einschreiben in die Register $D012 und Bit 7 von $D011.

2) Freigeben des Rasterzeileninterrupts durch Einschreiben von 1000 0001 in das Interrupt-enable-Register $D01A.

Der nächste Schritt betrifft die Bearbeitung des IRQ durch die CPU. Wie wir vorhin sahen, springt das Programm beim IRQ mittels eines indirekten Sprunges, der auf den Vektor 788/9 ($314/5) zugreift. Dieser Vektor muß nun auf die eigene Routine verbogen werden, also:

3) Vektor $314/5 auf die IRQ-Service-Routine richten.

Damit wären alle Vorbereitungen getroffen. Der Rest liegt nun ganz bei uns — beziehungsweise bei dem von uns zu schreibenden Service-Programm. Als Bild 48 finden Sie ein Flußdiagramm unseres Beispielprogrammes 7.

Gehen wir nun an die Realisierung. Zunächst also die Initialisierung, die wir bei $6000 (also durch SYS 24576 zu starten) beginnen lassen:

```
6000   SEI          Sperren von IRQs
Schritt 3:
6001   LDA #$28     LSB der IRQ-Routine
6003   STA 0314     in IRQ-Vektor-LSB
6006   LDA #$60     MSB der IRQ-Routine
6008   STA 0315     in IRQ-Vektor-MSB
Schritt 1:
600B   LDA #$F8     Rasterzeile, bei der das Textfen-
                    ster endet. Von da an soll der
                    Rahmen schwarz sein.
600D   STA D012     in Rasterzeilen-Register (LSB)
                    schreiben.
6010   LDA D011     Register mit dem msb des
                    Rasterzeilenzählers
6013   AND #$7F     0111 1111 löscht das Bit7
6015   STA D011     Zurückschreiben. Damit ist die
                    Rasterzeile, die den IRQ auslösen
                    soll, festgelegt.
Schritt 2:
6018   LDA #$81     1000 0001 wird nun
601A   STA D01A     ins IRQ-enable-Register
                    geschrieben, um den
                    Rasterzeilen-IRQ zuzulassen.
Festlegen einiger Startwerte:
601D   LDA #$00     Farbe schwarz
601F   STA D020     in Rahmen schreiben
6022   LDA #$04     Streifenbreite in
6024   STA 02       Merkregister schreiben.
6026   CLI          IRQ freigeben
6027   RTS          Ende der Initialisierung.
```

Von nun an laufen alle IRQs über unsere eigene Routine, die bei $6028 beginnt.

Zunächst müssen wir prüfen, ob die Unterbrechung vom VIC-II-Chip kommt oder vom CIA1:

```
6028   LDA D019     IRQ-Request-Register des VIC-II-
                    Chip (siehe Kapitel 51). Dort ist
                    Bit 7 gesetzt, wenn die Anforde-
                    rung vom VIC-II-Chip kam.
602B   STA D019      Zurückschreiben
602E   BMI 6037     Sprung, falls VIC-IRQ, sonst
                    CIA-IRQ.
Bearbeiten eines CIA-IRQ:
6030   LDA DC0D     Löschen des CIA1
                    Unterbrechungs-Kontrollregisters.
```

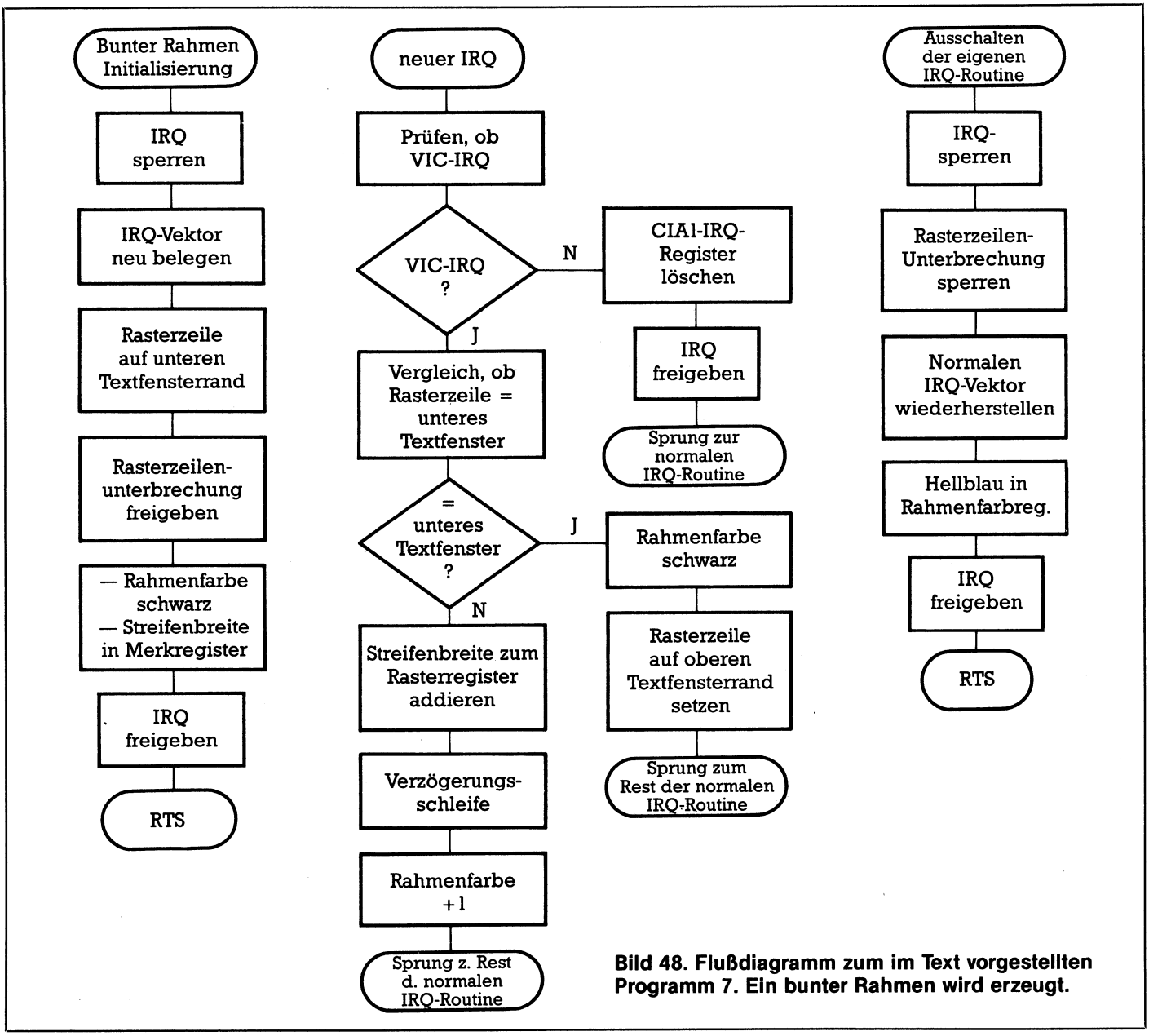


Bild 48. Flußdiagramm zum im Text vorgestellten Programm 7. Ein bunter Rahmen wird erzeugt.

6033	**CLI**	IRQ zulassen. Damit können innerhalb eines CIA- IRQ auch unsere VIC-IRQs geschehen.
6034	**JMP EA31**	Bearbeitung des CIA-IRQ durch die normale Routine.

Unser Programm für VIC-II-IRQs:

6037	**LDA D012**	Rasterzeilen-Register laden um festzustellen, welche Zeile den IRQ auslöste.
603A	**CMP #$F8**	Vergleich mit Ende des Textfensters.
603C	**BCS 604F**	Wenn unterhalb des Textfensters, Sprung.

Der folgende Programmteil ist wirksam, wenn der IRQ-Auslöser eine Zeile in Höhe des Textfensters war:

603E	**CLC**	Addition vorbereiten.
603F	**ADC 02**	Streifenbreite aus dem Merkregister addieren.
6041	**STA D012**	Neuen Wert in Rasterzeilen-Register schreiben.

Damit wird eine neue Rasterzeile als IRQ-Auslöser festgelegt, die um die Streifenbreite tiefer liegt als die vorhergegangene.

Es folgt eine kleine Verzögerungsschleife, die aber nur zum Experimentieren eingebaut wurde:

6044	**LDY #$03**	Schleifen-Startwert
6046	**DEY**	Herunterzählen
6047	**BNE 6046**	NEXT Y, bis Y=0.

Ändern der Rahmenfarbe bis zum nächsten Raster-IRQ:

6049	**INC D020**	Farbcode+1. Wenn Code im Rahmenfarbregister größer als 15 wird, fängt wieder Farbcode 0 an, weil die Bits 5-7 keine Funktion haben.

Abschließend erfolgt der Rücksprung in den Rest der normalen IRQ-Routine:

604C	**JMP EA81**	Siehe unsere Untersuchung der IRQ-Firmware.

Damit ist der Rahmen in Höhe des Textfensters behandelt. Es schließt sich nun der Teil an, der die Rahmenbereiche unter- und oberhalb bearbeitet:

604F	**LDA #$00**	Farbcode schwarz
6051	**STA D020**	in Rahmenfarb-Register.
6054	**LDA #$32**	Rasterzeile, bei der oben das Textfenster beginnt.
6056	**STA D012**	In Rasterzeilen-Register schreiben
6059	**JMP EA81**	Abschluß durch Sprung zum Ende der normalen IRQ- Routine.

Damit ist festgelegt, daß ober- und unterhalb des Textfensters die Rahmenfarbe schwarz wird.

Unsere eigene Routine ist jetzt abgeschlossen. Zum guten Ton gehört es, dem Benutzer auch die Möglichkeit zu öffnen, diese Routine wieder abzuschalten. Das erfolgt im letzten Programmteil, der mittels SYS24688 aktiviert werden kann:

605C	**SEI**	IRQ sperren
605D	**LDA #$00**	Raster-IRQ
605F	**STA D01A**	abschalten
6062	**LDA #$31**	IRQ-Vektor
6064	**STA 0314**	restaurieren
6067	**LDA #$EA**	auf den
6069	**STA 0315**	Normalwert.
606C	**LDA #$0E**	Farbcode hellblau
606E	**STA D020**	in Rahmenfarb-Register schreiben
6071	**CLI**	IRQ zulassen
6072	**RTS**	

Unser Programm ist komplett. Speichern Sie es bitte vor dem Starten ab. Nach dem SYS 24576 finden Sie einen hübschen bunten Rahmen vor, oberhalb und unterhalb des Textfensters ist er schwarz. Besonders gut — finde ich — sieht das Ganze aus, wenn man die Hintergrundfarbe des Textfensters auch auf Schwarz setzt. Das Programm erlaubt noch einige Experimente:

Durch POKE-Kommandos in die Speicherstelle 2 kann die aktuelle Streifenbreite variiert werden, durch POKEs in die Zelle 24645 der Startwert der Verzögerungsschleife. Probieren Sie's doch mal aus. Eine Erkenntnis werden Sie gewinnen: In der Unterbrechungs-Programmierung spielt die Zeit eine wichtige Rolle. Das zeigt sich auch, wenn man zum Beispiel Cursorbewegungen durchführt: Die Streifen fangen an zu wandern.

Weitere Möglichkeiten zum Experimentieren sind gegeben, wenn Sie die Rasterzeilen verändern, die den oberen und unteren Rand des Textfensters markieren:

Durch POKE 24661,Zahl verschieben Sie die obere, durch POKE 24635,X:POKE 24588,X die untere Rasterzeile, von der an alles schwarz ist. Wie schon vorhin erwähnt, habe ich im Programm diese Werte auf 50 beziehungsweise 248 fixiert, weil genau dort auf meinem Monitor das Textfenster liegt.

Mit diesem Beispiel und dem aus der Grafikserie sollte es Ihnen nun möglich sein, auch andere Unterbrechungsprogramme zu schreiben, die sich der Rasterzeilen-Unterbrechung per VIC-II-Chip bedienen. Eine Bemerkung sollte ich Ihnen noch auf den Weg Ihrer eigenen Versuche mitgeben: Der Elektronenstrahl, der über den Bildschirm saust und beim Erreichen des von uns bestimmten Rasterzeilenwertes zum Auslösen des IRQ führt, ist enorm schnell. Die Serviceprogramme dürfen deshalb nicht zu lang sein, sonst steht der nächste IRQ schon wieder an, bevor der vorangegangene bearbeitet ist.

63. Unterbrechungen mit den CIAs

Lassen Sie uns kurz rekapitulieren: Als primäre Unterbrechungsanforderer hatten wir drei Bausteine unseres Computers benannt, nämlich den VIC-II-Chip und die beiden CIA-Bausteine. CIA kommt von »Complex Interface Adapter« und ist die Bezeichnung für die beiden Ein- und Ausgabe-Bausteine, die den gesamten Verkehr zwischen dem zentralen Gehirn unseres C 64 und der Peripherie managen. Wir hatten bemerkt, daß ein CIA, der IRQ-CIA (Adressen von 56320 bis 56575), ausschließlich für die maskierbaren Unterbrechungen zuständig ist. Dazu gehören die 60mal pro Sekunde stattfindenden »Timer-Interrupts«, die die Cursorbehandlung, die TI$-Uhr, die Tastaturabfrage etc. bearbeiten. Der andere CIA, genannt NMI-CIA, (Adressenraum 56576-56831) ist nur für die nicht maskierbaren Unterbrechungen verantwortlich und wird bei normaler Nutzung des C 64 so gut wie nie eingesetzt. Ich gehe im folgenden davon aus, daß Sie keine RS232C-Schnittstelle in Ihren Computer eingesetzt haben. Sollte das aber der Fall sein, dann müßten Sie darauf achten, die folgenden Beispiele – die den NMI-CIA betreffen – ohne gleichzeitigen Betrieb dieser Schnittstelle anzuwenden, weil sich sonst Störungen ergeben könnten.

In Kapitel 52 haben wir uns ein Register (das Register 13, Interrupt-Kontrollregister) der CIAs schon genauer angesehen und auch die Unterschiede beider Bausteine festgestellt. Dort war dann die Rede von Timern, Echtzeituhren, Alarm-Funktionen etc. Was es damit auf sich hat und wie man diese Möglichkeiten nutzen kann, das soll nun unser Thema sein. Wir werden uns dazu alle Register der CIAs genauer ansehen, die für die von uns ausgewählten Unterbrechungsoptionen eine Rolle spielen. Dabei fallen einige unter den Tisch – das habe ich aber schon in Kapitel 52 angekündigt –, nämlich diejenigen, die mit dem Verkehr über den seriellen

Port, beziehungsweise über die RS232C-Schnittstelle, zu tun haben. Es bleibt dann anderen – kompetenteren – überlassen, darüber zu schreiben. Wie wäre es zum Beispiel mit Ihnen?

Auch so bleibt uns genug zu tun. In Tabelle 21 finden Sie zunächst eine Übersicht der von uns behandelten Register.

Sie sehen darin, daß jeder CIA über zwei sogenannte Timer (A und B) verfügt, sodann über die »Time of Day« (zu deutsch etwa »Tageszeit«) genannte Echtzeituhr mit vier Registern und schließlich noch über drei Kontrollregister, zu denen auch das schon erwähnte Register 13 gehört. Sehen wir uns zunächst die Timer an.

64. Die Timer der CIAs

Insgesamt verfügen wir über vier dieser Timer: Timer A und B im CIA1 und dasselbe nochmal im CIA2. Es handelt sich dabei um 16-Bit-Register, in die ein Startwert geschrieben werden kann, von dem an dann heruntergezählt wird. Jedesmal, wenn dann der Wert 0 unterschritten ist, gibt es für uns die Möglichkeit, bestimmte Ereignisse stattfinden zu lassen. Man kann diese Register unabhängig voneinander, aber auch kombiniert, benutzen. Ein Lesen des Registers liefert immer den momentan gerade aktuellen Wert. Ein Schreiben in das Register führt automatisch zum Festlegen eines Startwertes. Was an Optionen mit diesen Timern möglich ist, wird über Kontrollregister gesteuert. Das CRA (Register $ 0E) bezieht sich vor allem auf den Timer A, das CRB (Register $ 0F) auf Timer B. Die 16-Bit-Register werden – wie gewohnt – in der Form LSB/MSB betrieben. In den Timer A des CIA1 wird bei jedem I/O-Reset folgendes Wertepaar eingetragen:

56324	dezimal 37	LSB
56325	dezimal 64	MSB

Das entspricht einem Startwert von 16421. Im PAL-System hat der Quarz, der die Taktfrequenz bestimmt, eine Frequenz von 17.734472 MHz. Die Prozessorfrequenz errechnet sich daraus mittels Division durch 18 zu 985248.4 Hz (also etwas weniger als 1 MHz, was den europäischen C 64 langsamer macht als den amerikanischen, der etwas mehr als 1 MHz verwendet). Wenn mit dieser Geschwindigkeit der Timer heruntergezählt wird, erhält man alle $^1/_{60}$ Sekunden genau einen Unterlauf. Das ist der Weg, eine kontrollierte Zeitspanne durch den Timer zählen zu lassen. Sei X der gesuchte Startwert, der zu einer Spanne von T Sekunden führt, dann kann man X berechnen mittels:

$$X = 985248.4 * T$$

Der Integerwert von X ist dann in ein LSB und ein MSB zu teilen und in die Timer-Register einzutragen. Allerdings ergibt sich so eine natürliche Grenze. Die höchste durch 2 Byte darstellbare Zahl ist ja 65535. Wenn wir diesen Wert in den Timer schreiben, dann ist er alle 1/15 Sekunden auf 0 heruntergezählt. Für längere Zeiten ist aber vorgesorgt. Die beiden Timer A und B sind kombinierbar (wie, dazu kommen wir gleich noch) zu einem 32-Bit-Register. Die höchste Zahl X ist dann:

$$4\,294\,967\,296 = 2^{32}$$

Damit kann im Extremfall eine Herabzählzeit von 1 Stunde, 12 Minuten und zirka 40 Sekunden eingeplant werden, was für die meisten Zwecke ausreichen dürfte.

Möchten Sie also genau eine Sekunde Spielraum haben beim Herunterzählen, dann muß die Zahl 985248 als 4-Byte-Integer-Wert in die Speicher von Timer A und Timer B gebracht werden. Das führt dann zu den Werten 0, 15, 8, 160 (weil 985248 = 0*16777216 + 15*65536 + 8*256 + 160). 0 und 15 gelangen als MSB beziehungsweise LSB in Timer B (also Register 07 und 06), 8 und 160 sind MSB und LSB für den Timer A (Register 05 und 04). Sehen wir uns nun an, wie wir dem Computer sagen, was mit diesen Startwerten in den Timer-Registern geschehen soll. Die beiden Kontrollregister CRA und CRB beziehen sich weitgehend auf die gleichnamigen Timer. Im Bild 49 finden Sie das Register $0E, also CRA und in Bild 50 das andere Kontrollregister CRB ($0F):

Die Bedeutung der Bits 0 bis 4 ist – jeweils für den dazugehörigen Timer – identisch:

Bit 0 an dieser Stelle führt zum sofortigen Anhalten des Timers. 1 in diesem Bit startet das Herunterzählen.

Bits 1 und 2 Diese beiden Bits hängen mit dem externen Signalverkehr zusammen und werden von uns außer acht gelassen.

Bit 3 Ist dieses Bit = 1, dann ist der sogenannte »One Shot«-Betrieb des Timers aktiv. Das bedeutet, daß vom Startwert an heruntergezählt wird bis auf Null. Es findet nun das programmierte Ereignis statt (zum Beispiel ein IRQ). Anschließend wird der Startwert wieder eingeladen und der Timer gestoppt.

Im Gegensatz dazu läuft der »Continuous«-Betrieb, wenn das Bit den Wert 0 enthält. Dabei geschieht zunächst dasselbe wie beim One Shot Modus, der Timer wird aber nicht angehalten, sondern der ganze Vorgang wiederholt sich in einer Endlosschleife.

Bit 4 Ein Hineinschreiben einer 1 in dieses Bit erzeugt ein sofortiges Neuladen der Timer-Register mit dem Startwert. Dabei ist es gleich-

Register Nr. ($)	Adresse (dez.) CIA-1	CIA-2	Name	Funktion
04	56324	56580	TALO	TIMER A LSB
05	56325	56581	TAHI	TIMER A MSB
06	56326	56582	TBLO	TIMER B LSB
07	56327	56583	TBHI	TIMER B MSB
08	56328	56584	TOD10THS	$^1/_{10}$-Sekunden-Register
09	56329	56585	TODSEC	Sekunden-Register
0A	56330	56586	TODMIN	Minuten-Register
0B	56331	56587	TODHR	Stunden-Register, AM/PM-Flagge
0D	56333	56589	JCR	Unterbrechungs-Kontrollregister
0E	56334	56590	CRA	Kontrollregister A
0F	56335	56591	CRB	Kontrollregister B

Tabelle 21. Die wichtigen Register der beiden CIAs.

7	6	5	4	3	2	1	0
TODIN 50Hz 60 Hz	externer Signal-verkehr	in MODE	Force-load	ONE Shot / Continu-ous	externer Signal-verkehr		Start / Stop

Bild 49. Das Kontrollregister des Timers A.

7	6	5	4	3	2	1	0
ALARM	In MODE		Force-load	ONE Shot/ Continuous	externer Signal-verkehr		Start / Stop

Bild 50. Dasselbe für den Timer B.

Register Name	Nr.	7	6	5	4	3	2	1	0
TOD10THS	08	unbenutzt				$^1/_{10}$-Sekundenwert			
TODSEC	09	unbenutzt	Zehnerstelle Sekunden			Einerstelle Sekunden			
TODMIN	0A	unbenutzt	Zehnerstelle Minunten			Einerstelle Minuten			
TODHR	0B	AM/PM Flagge	unbenutzt		Zehnerstelle Stunden	Einerstelle Stunden			

Bild 51. Die Register der Echtzeituhren.

gültig, ob der Timer gerade läuft oder nicht. Schreibt man eine Null ein, hat das keine Wirkung.

Beim Lesen des Registers ist dieses Bit immer 0.

Zu diesem Bit und seiner Wirkung ist noch etwas zu sagen. Das Neuladen des Timers geschieht

- immer dann, wenn ein Unterlauf stattgefunden hat oder
- falls der Timer steht und in die Register ein Startwert geschrieben wird. Dabei ist der CIA so konstruiert, daß man kein zwangsweises Laden (also mit Bit 4 = 1) braucht, wenn man den Startwert in der Reihenfolge LSB MSB in die Register bringt.

Die Bits 5 bis 7 haben nun unterschiedliche Bedeutung im CRA und im CRB:

Register CRA ($0E)

Bit 5: Ist dieses Bit gleich Null, dann wird im Systemtakt gezählt. Den hatten wir vorhin zur Zeitberechnung schon verwendet. Wenn das Bit auf 1 gesetzt ist, zählt der Timer externe Signale.

Bit 6: Spielt für den Signalverkehr über den seriellen Port eine Rolle und soll uns hier nicht weiter beschäftigen.

Bit 7: Damit steuert man nicht den Timer A, sondern dieses Bit bezieht sich auf die gleich noch zu behandelnde Echtzeituhr.

Register CRB ($0F)

Die Bits 5 und 6 sind hier im Zusammenhang von Bedeutung. Es gibt vier Kombinationsmöglichkeiten:

Bit 6 - Bit 5

0 - 0 Systemtakt heruntergezählt.

0 - 1 Der Timer B wird durch externe Signale heruntergezählt.

1 - 0 Der Timer B zählt die Unterläufe von Timer A. Das ist der vorhin erwähnte Punkt, der beide Timer kombiniert zum 32-Bit-Zähler. Man kann also im Extremfall 65536 mal 65536 Takte zählen lassen.

1 - 1 Auch in diesem Fall zählt Timer B die Unterläufe von Timer A. Er tut das aber nur, wenn ein bestimmtes externes Signal vorhanden ist.

Bit 7: Auch beim Register CRB steuert dieses Bit bestimmte Möglichkeiten der Echtzeituhr. Deshalb haben Sie noch ein wenig Geduld, bis wir diese Uhr behandeln.

Wir kennen uns nun ganz gut aus, wie wir mit den Timern umzugehen haben. Unser Wissen soll in einem kleinen Test erprobt werden. Dazu bedienen wir uns des 1/60 Sekunden IRQ. Wir verändern diese regelmäßige Unterbrechung derart, daß sie nur noch einmal in der Sekunde geschieht. Welche Zahlen dazu in ein 32-Bit-Register gepackt werden müssen, haben wir schon vorhin berechnet. Jeweils in der Reihenfolge LSB/MSB müssen wir sie einschreiben und vorher die Timer anhalten, indem die Bits 0 der Kontrollregister CRA und CRB auf 0 gesetzt werden. Nach dem Einschreiben und Starten der beiden Timer müssen folgende Bitmuster in CRA und CRB stehen:

CRA

Bit 0 = 1 Start Timer A
Bit 3 = 0 Dauerlauf
Bit 5 = 0 Systemtakt

CRB

Bit 0 = 1 Start Timer B
Bit 3 = 0 Dauerlauf
Bit 5 = 0
Bit 6 = 1 Timer B zählt Unterläufe von Timer A.

Bevor wir die Timer starten, muß auch noch das Interrupt-Kontrollregister verändert werden (das hatten wir uns in Kapitel 52 genauer angesehen). Bislang erzeugt ein Unterlauf des Timers A eine Unterbrechung. Wir möchten aber, daß der Timer B (damit wir das 32-Bit-Register voll ausnutzen) der Auslöser ist. Dazu muß Bit 0 des ICR gelöscht und stattdessen Bit 1 gesetzt werden.

Im Programm »Timer-Test« (siehe Listing 8 und 9) ist all das realisiert. Mit SYS 49152 gestartet, zeigt sich sofort ein deutlich verlangsamter Cursor. Noch langsamer kann alles werden, indem Sie höhere Werte in die Timer-Register schreiben.

```
programm : prg.timer-testc000 c051
-----------------------------------

c000 : 78 ad 0e dc 29 fe 8d 0e  4b
c008 : dc ad 0f dc 29 fe 8d 0f  f9
c010 : dc a9 0f 8d 06 dc a9 00  24
c018 : 8d 07 dc a9 a0 8d 04 dc  d5
c020 : a9 08 8d 05 dc a9 1f 8d  84
c028 : 0d dc a9 82 8d 0d dc ad  6e
c030 : 0e dc 29 d7 8d 0e dc ad  0a
c038 : 0f dc 29 d7 8d 0f dc ad  1b
c040 : 0e dc 09 01 8d 0e dc ad  37
c048 : 0f dc 09 41 8d 0f dc 58  a5
c050 : 60 ff 00 ff 00 ff 00 ff  b0
```

Listing 8. Programm Timer-Test, ein Beispiel für die Anwendung eines 32-Bit-Timers.

```
PASS 1


PASS 2
7000            0823 ;
7000            084C ;****************************************
7000            0375 ;*                                      *
7000            089E ;*              TIMER-TEST              *
7000            08C7 ;*                                      *
7000            08F0 ;* TIMER A UND B DES CIA1 WERDEN SO    *
7000            0919 ;* GESCHALTET, DASS NUR NOCH 1 MAL     *
7000            0942 ;* PRO SEKUNDE DER TIMER-IRQ AUFTRITT *
7000            096B ;*                                      *
7000            0994 ;*    HEIMO PONNATH  HAMBURG  1985     *
7000            09BD ;*                                      *
7000            09E6 ;****************************************
7000            09E9 ;
7000            09EC ;
C000            09F9                .BA $C000
C000            09FE                .OS
C000            0A01 ;
C000            0A27 ;+++ BENUTZTE ADRESSEN DES CIA 1 +++
C000            0A2A ;
C000            0A3A TALO           .DE $DC04
C000            0A4A TAHI           .DE $DC05
C000            0A5A TBLO           .DE $DC06
C000            0A6A TBHI           .DE $DC07
C000            0A79 ICR            .DE $DC0D
C000            0A88 CRA            .DE $DC0E
C000            0A97 CRB            .DE $DC0F
C000            0A9A ;
C000            0AC3 ;+++ EINSCHALTEN DES 1 SEKUNDEN IRQ +++
C000            0AC6 ;
C000 78         0AE5 START          SEI                     ;SPERREN ALLER IRQS
C001            0AE8 ;
C001 AD 0E DC   0AF2                LDA CRA
C004 29 FE      0B03                AND #%11111110
C006 8D 0E DC   0B1B                STA CRA                 ;STOP TIMER A
C009 AD 0F DC   0B25                LDA CRB
C00C 29 FE      0B36                AND #%11111110
C00E 8D 0F DC   0B4E                STA CRB                 ;STOP TIMER B
C011            0B51 ;
C011 A9 0F      0B6F                LDA #15                 ;NEUER STARTWERT IN
C013 8D 06 DC   0B8A                STA TBLO                ;32-BIT-REGISTER
C016 A9 00      0B94                LDA #00
C018 8D 07 DC   0B9F                STA TBHI
C01B A9 A0      0BAA                LDA #160
C01D 8D 04 DC   0BB5                STA TALO
C020 A9 08      0BBF                LDA #08
C022 8D 05 DC   0BCA                STA TAHI
C025            0BCD ;
C025 A9 1F      0BDE                LDA #%00011111
C027 8D 0D DC   0BFB                STA ICR                 ;ALLE IRQ VERBOTEN
C02A A9 82      0C0C                LDA #%10000010
C02C 8D 0D DC   0C27                STA ICR                 ;NUR TIMER B IRQ
C02F            0C2A ;
C02F AD 0E DC   0C34                LDA CRA
C032 29 D7      0C45                AND #%11010111
C034 8D 0E DC   0C61                STA CRA                 ;BITS 3 UND 5 = 0
C037            0C64 ;
C037 AD 0F DC   0C6E                LDA CRB
C03A 29 D7      0C7F                AND #%11010111
C03C 8D 0F DC   0C8F                STA CRB                 ;DITO
C03F            0C92 ;
C03F AD 0E DC   0C9C                LDA CRA
C042 09 01      0CAD                ORA #%00000001
C044 8D 0E DC   0CC6                STA CRA                 ;TIMER A START
C047            0CC9 ;
C047 AD 0F DC   0CD3                LDA CRB
C04A 09 41      0CE4                ORA #%01000001
C04C 8D 0F DC   0D01                STA CRB                 ;TIMER B START MIT
C04F            0D24 ;                                      TIMER A UNTERLAUF
C04F            0D27 ;
C04F 58         0D3D                CLI                     ;IRQS FREIGEBEN
C050            0D40 ;
C050 60         0D46                RTS
C051            0D49 ;
C051            0D4F                .EN
```

Listing 9. Der Quelltext zum Timer-Set.

Den Normalzustand stellen Sie einfach durch Drücken der RUN/STOP- und der RESTORE-Tasten her. Dabei wird ja – wie Sie aus dem letzten Kapitel her wissen, auch ein I/O-Reset ausgeführt, der den Ausgangszustand wiederherstellt.

Die Verlängerung des IRQ-Zyklus hat übrigens noch einen sinnvollen Nebeneffekt. Je seltener ein laufendes Programm unterbrochen wird, desto schneller wird es mit seinen Jobs fertig. Das kann man immer dann tun – im Extremfall sogar den IRQ ganz ausschalten – wenn man die Möglichkeiten, die der Computer während des normalen IRQ anbietet, nur selten oder aber gar nicht braucht.

65. Die Echtzeituhren

Wir kennen nun fünf Uhren in unserem Computer: Die vier Timer (jeweils A und B im CIA1 und CIA2), die wir, weil wir die Impulszahlen in Zeiteinheiten umrechnen können, zur Zeitmessung einsetzen könnten und die im Basic verfügbare Uhr TI$, die aber – wie wir nun wissen – lediglich die Umsetzung des Timers A im CIA1 in ein bequemer handhabbares Software-Instrument ist. Zudem ist die Ganggenauigkeit dieser Uhr recht gering. Schon einige Kassettenoperationen genügen, sie völlig aus dem Takt zu bringen.

Um so mehr verwundert es, daß zwei hervorragende Echtzeituhren im Commodore 64 so gut wie nie benutzt werden, ja nicht einmal in irgendeiner Weise softwaremäßig unterstützt werden. Vielleicht ist das ein bißchen zuviel »mehr sein als scheinen«, was Commodore da betreibt, wenn man bedenkt, welche verborgenen Schätze da alle zutage gefördert werden können (man denke nur an die hochauflösende Grafik) bei genauer Untersuchung des Computers.

Jeder der beiden CIAs verfügt über solch eine Uhr, die direkt von der Netzfrequenz getaktet wird. Die Zählung der Zeit geschieht in vier Registern (Register $08 bis $0B), die in Bild 51 gezeigt sind.

Vielleicht fällt Ihnen etwas auf, wenn Sie sich diese vier Byte mal genauer ansehen: Die Speicherung geschieht in Form von Einer- und Zehnerstellen. Das kann also weder im Binärformat noch als ASCII-Zeichen funktionieren. Hier werden die Ziffern als BCD-Zahlen abgelegt. In Kapitel 13 wurde dieses »binary coded decimal«-Format erklärt. Das ist lange her und soll deshalb hier nochmal vorgestellt werden, damit alle wissen, wovon die Rede ist.

In dieser Zahlendarstellung wird jede Dezimalstelle einer Zahl gesondert in eine Binärzahl umgewandelt. Dann ergibt sich der folgende Zusammenhang:

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

Das war's! Die anderen möglichen Binärkombinationen (also zum Beispiel 1010 etc.) werden nicht benutzt. Die Zahl 25 beispielsweise lautet im BCD-Format:

```
0010  0101
  ↑     ↑
  2     5
```

Jetzt ist es Ihnen sicherlich verständlich, warum für die Sekunden- und Minuten-Zehnerstellen nicht mehr als drei Bit reserviert wurden: größer als 6 wird die Zehnerstelle nicht.

Zum Stundenregister TODHR ist aber noch etwas zu sagen: Dort ist nur ein Bit reserviert für die Stunden-Zehnerstelle. Die Uhr läuft nicht bis 24 Uhr, sondern lediglich bis 12 Uhr. Zur Unterscheidung, ob vor- oder nachmittags gemeint ist, dient das Bit 7. Dieses sogenannte AM/PM-Flag ist orientiert an der angelsächsischen Gewohnheit, zum Beispiel für 16 Uhr den Ausdruck 4 PM zu verwenden. PM kommt vom lateinischen »post meridiem«, was übersetzt heißt »nach dem Mittag«, wohingegen AM steht für »ante meridiem«, also »vor dem Mittag«. Meint man nun AM, dann muß diese Flagge auf 0, bei PM aber auf 1 gesetzt sein.

Beim Stellen der Uhren sollte eine Reihenfolge eingehalten werden. Sobald nämlich in das Stundenregister geschrieben wird, hält die Zählung automatisch an. Man kann nun die anderen Werte in die Register schreiben. Den Startschuß liefert das Schreiben in das Register TOD10TH: von nun an tickt die Uhr wieder.

Ähnlich funktioniert das Lesen der Uhrzeit. Sobald das Stundenregister gelesen wird, führt das zum Anhalten der Uhr, so daß die restlichen Register reibungslos auslesbar sind. Wieder ist es das Zehntelsekundenregister, das beim Auslesen ein Weiterlaufen der Uhr bewirkt. Aber, so werden Sie bemerken, wenn der Auslesevorgang eine bestimmte Zeit beansprucht, führt das zu Verzögerungen? Die Lösung ist, daß der gesamte Inhalt der vier Register gleichzeitig mit dem Auslesen des Stundenwertes in einen internen Speicher transferiert wird und dort weiterläuft. Nach dem Lesen des TOD10TH kommt der aktuelle Wert zurück in die Register und dieser wird weitergezählt.

Nun wird es höchste Zeit, daß wir uns die beiden Bits im CRA und im CRB ansehen, die wir vorhin bei der Timer-Behandlung links liegen ließen. Bit 7 im CRA kündigt der Echtzeituhr an, welche Netzfrequenz zu erwarten ist. Eine 1 an dieser Stelle steht für 50 Hz, eine 0 für 60 Hz. Unser Stromnetz in Deutschland liefert einen Wechselstrom mit 50 Hz, weshalb wir dann dort die 1 setzen sollten. Da gibt es ein kleines Problem: Beim I/O-Reset, der durch Drücken der RUN/STOP- und der RESTORE-Tasten zusammen ausgelöst wird, schreibt der Computer immer den amerikanischen Wert für 60 Hz in dieses Bit. Dann geht die Uhr aber empfindlich nach. Man muß also einen Weg finden, der erlaubt, dort in diesem Fall wieder eine 1 einzuschreiben. Das ist durch eine eigene NMI-Routine möglich. Sie sehen schon, der Weg zur Nutzung dieser verlockenden Uhren ist ziemlich dornenreich!

Noch interessanter ist das Bit 7 im CRB. Das Setzen der Uhrzeit ist nämlich nur möglich, wenn dieses Bit den Inhalt 0 hat. Was geschieht, wenn dort eine 1 steht? Dann bestimmt man nicht die aktuelle Uhrzeit, sondern man stellt einen Wecker (das ist die Alarmzeit). Das geschieht nach dem Setzen dieses Bits genauso wie vorhin das Einschreiben der Uhrzeit (also erstaunlicherweise auch in genau dieselben Register!). Im Unterschied dazu ist allerdings ein Lesen der Alarmzeit nicht möglich – das ergibt immer die aktuelle Uhrzeit. Man muß für diesen Fall die Weckzeit irgendwo abspeichern und bei Bedarf dann von dort lesen.

Weil man ja meistens nach dem Erreichen der Alarmzeit irgendeine Reaktion erwartet, ist im ICR (also dem Unterbrechungskontrollregister 13) jedes CIAs noch ein Bit reserviert – das Bit 2 —, mit dessen Hilfe der Alarm per IRQ oder NMI wie auch immer geartet losbrechen kann. Der Phantasie sind hier nur wenige Grenzen gesetzt. Wie man mit diesem ICR umgeht, haben wir schon besprochen.

Damit sind wir durch die Eigenheiten der CIAs durch. Man braucht tatsächlich keine Scheu zu haben, diese Echtzeituhren zu nutzen. Lediglich die Uhr im CIA1 wird manchmal verwendet, einen bestimmten Wert für die Zufallszahlenerzeugung zu generieren. Aber das sollte einer eigenen Uhren-

Routine nicht in die Quere kommen. Solch eine Echtzeituhr finden Sie in Listing 10 und 11.

Durch SYS49152 aktivieren Sie die Uhr, die Sie mit SYS49261 auch wieder abschalten können. Durch ein USR-Kommando A=USR (String) stellen Sie die Startzeit ein. String kann dabei eine Stringvariable sein oder auch direkt ein String der Form »HHMMSST« (also Stunden, Minuten, Sekunden, Zehntelsekunden). In A steht eine 0, wenn kein Fehler, aber eine -1, wenn ein Fehler aufgetreten ist. Das Lesen der Uhr erfolgt über ein zweites USR-Kommando:

PRINTUSR(Zahl). Dabei kann Zahl eine beliebige Zahl oder Variable sein. Eine Alarmzeit ist ebenfalls einstellbar durch ein USR-Kommando, in dem vor der Zeiteingabe noch ein Buchstabe steht. Beispielsweise stellt A=USR(»A1200000«) einen Wecker auf 12 Uhr. Der Alarm im Programm läßt den Bildschirmrahmen blinken. Abstellen kann man das durch Auslösen eines RESTORE-NMI (also RUN/STOP und RESTORE). Sollten Sie vor dem eingestellten Alarm mal solch einen NMI auslösen, dann muß die Alarmzeit neu gestellt werden. Als Basis für dieses Programm diente ein Listing aus dem schon oft erwähnten Buch von Babel/Krause/Dripke »Das Interface Age Systemhandbuch zum Commodore 64«.

Die Unterbrechungs-Programmierung ist damit abgeschlossen – ebenso dieser Kurs, der als Einführung in die Assembler-Alchimie nun alle Geheimnisse der Kunst aufgedeckt hat. In den letzten Kapiteln sind wir schon in die Meistergrade der Zunft aufgestiegen. Vielleicht ging es manchem etwas zu schnell? Dann wird Ihnen der Kurs »Von Basic zu Assembler« eine Hilfe sein, der behutsam und mit vielen an Basic angelehnten Beispielen die nötige Programmierpraxis vermitteln wird (ab 64'er, Ausgabe 1/86). So wie die Segler sich oft »Mast- und Schotbruch« wünschen, verabschiede ich mich, indem ich Ihnen viele grandiose Abstürze wünsche.

(Heimo Ponnath/gk)

```
 programm : obj.alarmuhr  c000 c18d
-----------------------------------

c000 : a9 8e 8d 11 03 a9 c0 8d  11
c008 : 12 03 a9 1d 8d 18 03 a9  a3
c010 : c0 8d 19 03 ad 0e dd 09  12
c018 : 80 8d 0e dd 60 48 8a 48  a1
c020 : 98 48 a9 7f 8d 0d dd ac  49
c028 : 0d dd 10 06 4c 6a c1 4c  a0
c030 : 72 fe 20 bc f6 20 e1 ff  b9
c038 : d0 f5 a2 04 bd 2f fd 9d  b4
c040 : 13 03 ca d0 f7 a2 1a bd  1a
c048 : 35 fd 9d 19 03 ca d0 f7  c0
c050 : a9 7f 8d 0d dc 8d 0d dd  e8
c058 : 8d 00 dc a9 08 8d 0e dc  30
c060 : a9 88 8d 0e dd a9 08 20  fe
c068 : b6 fd 4c 6c fe a9 48 8d  37
c070 : 11 03 a9 b2 8d 12 03 78  2a
c078 : a9 47 8d 18 03 a9 fe 8d  c0
c080 : 19 03 a9 31 8d 14 03 a9  84
c088 : ea 8d 15 03 58 60 24 0d  12
c090 : 30 03 4c 20 c1 20 82 b7  f0
c098 : c0 07 d0 40 ad 0f dd 29  35
c0a0 : 7f 8d 0f dd a0 00 a9 24  5e
c0a8 : 20 fe c0 d0 02 a9 24 c9  23
c0b0 : 13 90 07 f8 38 e9 12 d8  b9
c0b8 : 09 80 8d 0b dd 20 fc c0  1a
c0c0 : 8d 0a dd 20 fc c0 8d 09  ec
c0c8 : dd 20 66 c1 8d 08 dd a9  6b
c0d0 : 00 4c 3c bc 68 68 68 68  d9
c0d8 : a9 ff d0 f5 c0 08 d0 f8  f5
c0e0 : ad 0f dd 09 80 8d 0f dd  1a
c0e8 : a9 84 8d 0d dd a9 3c 85  ff
c0f0 : 04 85 02 a9 ff 85 03 a0  e6
c0f8 : 01 4c a6 c0 a9 60 85 24  dd
c100 : 20 13 c1 0a 0a 0a 0a 85  80
c108 : 25 20 13 c1 05 25 c5 24  13
c110 : b0 c4 60 b1 22 38 e9 30  5d
c118 : 90 ba c9 0a b0 b6 c8 60  5e
c120 : a9 07 20 7d b4 a0 00 ad  b0
c128 : 0b dd 08 29 1f c9 12 d0  73
c130 : 02 a9 00 28 10 05 f8 18  49
c138 : 69 12 d8 20 55 c1 ad 0a  13
c140 : dd 20 55 c1 ad 09 dd 20  96
c148 : 55 c1 ad 08 dd 20 60 c1  ce
c150 : 68 68 4c ca b4 48 4a 4a  a4
c158 : 4a 4a 20 60 c1 68 29 0f  fe
c160 : 09 30 91 62 c8 60 20 13  68
c168 : c1 60 a9 77 8d 14 03 a9  8b
c170 : c1 8d 15 03 4c bc fe c6  d2
c178 : 02 f0 03 4c 31 ea a5 04  46
c180 : 85 02 ad 20 d0 45 03 8d  d4
c188 : 20 d0 4c 31 ea 00 ff 00  f8
```

Listing 10. Eine Echtzeituhr.

```
PASS 2
7000           0823 ;
7000           084C ;***************************************
7000           0875 ;*                                     *
7000           089E ;*  ECHTZEITUHR MIT ALARMFUNKTION      *
7000           08C7 ;*                                     *
7000           08F0 ;*  LAEUFT MIT DEM NMI-CIA             *
7000           0919 ;*  IN VERBINDUNG MIT DEM IRQ FUER     *
7000           0942 ;*  DEN ALARM                          *
7000           096B ;*                                     *
7000           0994 ;*    HEIMO PONNATH  HAMBURG  1985     *
7000           09BD ;*                                     *
7000           09E6 ;*  (TEILWEISE WURDE EIN PROGRAMM AUS  *
7000           0A0F ;*  DEM INTERFACE AGE SYSTEMHANDBUCH   *
7000           0A38 ;*  ZUM COMMODORE 64 , SEITE 114       *
7000           0A61 ;*  ALS BASIS VERWENDET )              *
7000           0A8A ;*                                     *
7000           0AB3 ;***************************************
7000           0ABC ;
C000           0AC2                .BA $C000
C000           0AC8                .OS
C000           0ACB ;
C000           0AF4 ;********** ZEROPAGE-LABELS ***********
C000           0AF7 ;
C000           0B18 VERZ           .DE $02             ;AKTUELLE VERZOEG.
C000           0B38 FARB           .DE $03             ;WERT FUER RAHMEN
C000           0B57 ;                  EOR-OPERATION
C000           0B78 VORW           .DE $04             ;VERZOEGERUNGSWERT
C000           0B9B VALTYP         .DE $0D             ;INHALT:FF=STR 0=N
C000           0B9E ;
C000           0BAD INDEX          .DE $22
C000           0BC6 INDEX3         .DE $24             ;POINTER
C000           0BD6 INDEX4         .DE $25
C000           0BF5 FAC1           .DE $62             ;1.MANTISSENBYTE
C000           0BF8 ;
C000           0C21 ;********** LABELS PAGES 3 ************
C000           0C24 ;
C000           0C44 USRADDL        .DE $0311           ;USR-POINTER
C000           0C57 USRADDH        .DE $0312
C000           0C5A ;
C000           0C6A FREI           .DE $0313
C000           0C6D ;
C000           0C8A IRQVL          .DE $0314           ;IRQ-VEKTOR
C000           0C9B IRQVH          .DE $0315
C000           0C9E ;
C000           0CBC NMINVL         .DE $0318           ;NMI-VEKTOR
C000           0CCE NMINVH         .DE $0319
C000           0CD1 ;
C000           0CFA ;********** LABELS INTERPRETER ********
C000           0CFD ;
C000           0D23 ILLQUERR       .DE $B248           ;ILLEGAL QUANTITY
C000           0D4B ;             ERROR=NORMALWERT USR-VEKTOR
C000           0D4E ;
C000           0D70 STRINI8        .DE $B47D           ;SPEICHERPLATZ
C000           0D99 ;             PRUEFEN,STRINGPOINTER SETZEN
C000           0DBF STRLIT67       .DE $B4CA           ;REST DER STRING-
C000           0DD8 ;             LESE-ROUTINE
C000           0DF7 LEN1           .DE $B782           ;STRINGLAENGE
C000           0E11 ;             IN Y-REGISTER
C000           0E32 ACTOFC         .DE $BC3C           ;AKKU NACH FAC
C000           0E35 ;
C000           0E5E ;********** LABELS VIC-II-CHIP ********
C000           0E61 ;
C000           0E7E RAND           .DE $D020           ;RAHMENFARBE
C000           0E81 ;
C000           0EAA ;********** LABELS CIA-BAUSTEINE ******
C000           0EAD ;
C000           0ECA CIA1           .DE $DC00           ;START CIA-1
C000           0EEC ICR1           .DE $DC0D           ;IRQ-KONTROLLREG.
C000           0F0E CRA1           .DE $DC0E           ;TIMER-A KONTRREG
C000           0F11 ;
C000           0F34 TOD10TH2       .DE $DD08           ;1/10 SEKUNDEN
C000           0F51 TODSEC2        .DE $DD09           ;SEKUNDEN
C000           0F6D TODMIN2        .DE $DD0A           ;MINUTEN
C000           0F8F TODHR          .DE $DD0B           ;STUNDEN + AM/PM
C000           0FB1 ICR2           .DE $DD0D           ;NMI-KONTROLLREG.
C000           0FD3 CRA2           .DE $DD0E           ;TIMER-A KONTRREG
C000           0FF5 CRB2           .DE $DD0F           ;TIMER-B KONTRREG
C000           0FF8 ;
C000           0FFB ;
C000           1024 ;********** LABELS OBERES ROM **********
C000           1027 ;
C000           1045 NORM           .DE $EA31           ;NORMALER IRQ
C000           1048 ;
C000           106B TASTFLAG       .DE $F6BC           ;TEIL DER NMI-
C000           108E ;              ROUTINE (KEIN MODUL)
C000           10AD VECTAB         .DE $FD2F           ;TABELLE DER
C000           10C8 ;              ROM-VEKTOREN
C000           10E9 VECTAB7        .DE $FD35           ;MSB DES NMI-
C000           110E ;              VEKTORS IN DER TABELLE
C000           1132 IORESET19      .DE $FDB6           ;I/O-RESET:BEI
C000           1157 ;              SETZEN DES CRA IRQ-CIA
C000           1178 NMIXCT16       .DE $FE6C           ;NMI-ROUTINE AB
C000           119D ;              SCREEN-EDITOR-RESET
C000           11C1 NMIRS232       .DE $FE72           ;NMI-ROUTINE AB
C000           11DE ;              RS232-HANDLING
C000           11FF NMIEND         .DE $FEBC           ;ENDE DER NMI-
C000           1215 ;              ROUTINE
C000           1237 STOP           .DE $FFE1           ;KERNAL STOP SPRG
C000           1254 ;              NACH JMP($328)
C000           1257 ;
C000           125A ;
C000           1283 ;********** AKTIVIEREN ****************
C000           1286 ;
C000 A9 8E     12A2 INIT           LDA #L,USR          ;USR-VEKTOR
C002 8D 11 03  12B7                STA USRADDL         ;LADEN
C005 A9 C0     12C4                LDA #H,USR
C007 8D 12 03  12D2                STA USRADDH
C00A           12D5 ;
C00A A9 1D     12F2                LDA #L,NMI          ;NMI-VEKTOR MIT
C00C 8D 18 03  130D                STA NMINVL          ;STARTADRESSE
C00F A9 C0     1327                LDA #H,NMI          ;DER EIGENEN
C011 8D 19 03  1345                STA NMINVH          ;NMI-ROUTINE LAD
C014           1348 ;
C014 AD 0E DD  1365                LDA CRA2            ;BIT7 CRA SETZEN:
C017 09 80     137C                ORA #%80            ;%1000 0000
C019 8D 0E DD  1397                STA CRA2            ;NETZFREQ.=50HZ
C01C           139A ;
C01C 60        13A0                RTS
C01D           13A3 ;
C01D           13CC ;********** EIGENE NMI-ROUTINE ********
C01D           13CF ;
C01D 48        13EF NMI            PHA                 ;ANFANG NORMALE NMI-R.
```

Listing 11. Der Quelltext zur Echtzeituhr.

```
C01E 8A       1406              TXA                 ;REGISTER RETTEN
C01F 48       140C              PHA
C020 98       1412              TYA
C021 48       1418              PHA
C022          141B ;
C022 A9 7F    1439              LDA #$7F            ;SPERREN ALLER NMI
C024 8D 0D DD 1444              STA ICR2
C027          1447 ;
C027 AC 0D DD 1462              LDY ICR2            ;PRUEFEN OB NMI
C02A 10 06    1481              BPL RESTNMI         ;VOM CIA2 KOMMT.
C02C          14A4 ;                 WENN NEIN=SPRUNG
C02C 4C 6A C1 14C1              JMP ALARM           ;WENN JA, ALARM
C02F 4C 72 FE 14E1 CIANMI       JMP NMIRS232        ;REST DER
C032          1508 ;                 NORMALEN NMI-ROUTINE
C032          150B ;
C032          1534 ;***** EIGENE RESTORE-NMI-ROUTINE *****
C032          1537 ;
C032          155F ;   DIE MODULPRUEFUNG WIRD AUSGELASSEN
C032          1562 ;
C032 20 BC F6 1587 RESTNMI      JSR TASTFLAG        ;TEIL DER NMI-
C035 20 E1 FF 15A5              JSR STOP            ;ROUTINE ZUR STOP-
C038 D0 F5    15C2              BNE CIANMI          ;TASTEN-ABFRAGE
C03A          15C5 ;
C03A A2 04    15E3              LDX #$04            ;IRQ UND BRK VEKT.
C03C BD 2F FD 1606 UMLAD1       LDA VECTAB,X        ;RESTAURIEREN
C03F 9D 13 03 1613              STA FREI,X
C042 CA       1619              DEX
C043 D0 F7    1626              BNE UMLAD1
C045          1629 ;
C045          1651 ;   DER NMI-VEKTOR WIRD UEBERSPRUNGEN
C045          1654 ;
C045 A2 1A    1671              LDX #$1A            ;RESTAURIEREN DER
C047 BD 35 FD 1693 UMLAD2       LDA VECTAB7,X       ;RESTLICHEN
C04A 9D 19 03 16AC              STA NMINVH,X        ;VEKTOREN
C04D CA       16B2              DEX
C04E D0 F7    16BF              BNE UMLAD2
C050          16C2 ;
C050          16E3 ;  ZUNAECHST NORMALER I/O-RESET
C050          16E6 ;
C050 A9 7F    16FD              LDA #$7F            ;= 0111 111
C052 8D 0D DC 171B              STA ICR1            ;SPERREN ALLER IRQ
C055 8D 0D DD 1739              STA ICR2            ;SPERREN ALLER NMI
C058 8D 00 DC 1753              STA CIA1            ;DATENREGISTER
C05B          177A ;                 PORT A AUF NORMALWERT
C05B A9 08    1791              LDA #$08            ;=0000 1000
C05D 8D 0E DC 17AD              STA CRA1            ;TIMER A IM CIA1
C060          17B0 ;
C060          17D2 ;  ERSATZ FUER BELEGUNG DES CRA2
C060          17D5 ;
C060 A9 88    17EC              LDA #$88            ;=1000 1000
C062 8D 0E DD 1809              STA CRA2            ;TIMER A IM CIA2:
C065          1829 ;                 BIT 0 AUF STOP
C065          184F ;                 BIT 3 AUF EINZELLAUF
C065          1875 ;                 BIT 5 SYSTEMTAKT EIN
C065          189D ;                 BIT 7 ECHTZEITUHR=50HZ
C065          18A0 ;
C065          18C0 ;  REST DES NORMALEN I/O-RESET
C065          18C3 ;
C065          18C6 ;
C065 A9 08    18DE              LDA #$08            ;= 0000 1000
C067 20 B6 FD 18EE              JSR IORESET19
C06A          18F1 ;
C06A          1919 ;  REST DER NORMALEN RESTORE-NMI-ROUT.
C06A          191C ;
C06A 4C 6C FE 193A              JMP NMIXCT16        ;EINSPRUNG BEI
C06D          195F ;                 SCREEN EDITOR RESET
C06D          1962 ;
C06D          198B ;**** ABSCHALTEN DER TIME OF DAY UHR **
C06D          198E ;
C06D          19AD ;            DURCH SYS-KOMMANDO
C06D          19B0 ;
C06D A9 48    19D1 AUS          LDA #L,ILLQUERR     ;USR-VEKTOR
C06F 8D 11 03 19EF              STA USRADDL         ;AUF NORMALWERT
C072 A9 B2    1A01              LDA #H,ILLQUERR
C074 8D 12 03 1A0F              STA USRADDH
C077          1A12 ;
C077 78       1A18              SEI
C078 A9 47    1A35              LDA #$47            ;RESTAURIEREN DES
C07A 8D 18 03 1A4F              STA NMINVL          ;NMI-VEKTORS
C07D A9 FE    1A5A              LDA #$FE
C07F 8D 19 03 1A67              STA NMINVH
C082          1A6A ;
C082 A9 31    1A87              LDA #L,NORM         ;RESTAURIEREN
C084 8D 14 03 1AA4              STA IRQVL           ;DES IRQ-VEKTORS
C087 A9 EA    1AB2              LDA #H,NORM
C089 8D 15 03 1ABE              STA IRQVH
C08C          1AC1 ;
C08C 58       1AC7              CLI
C08D 60       1ACD              RTS
C08E          1AD0 ;
C08E          1AF9 ;**** DURCH USR AUFRUFBARE ROUTINE ****
C08E          1AFC ;
C08E 24 0D    1B1D USR          BIT VALTYP          ;WELCHER TYP VON
C090          1B44 ;                 VARIABLEN LIEGT VOR ?
C090 30 03    1B5F              BMI STRING          ;WENN STRING,
C092          1B83 ;                 DANN UEBERSPRINGEN
C092 4C 20 C1 1B9F              JMP ZAHLVAR         ;SONST SPRUNG
C095          1BA2 ;
C095          1BCB ;****** STELLEN DER ECHTZEITUHR *******
C095          1BEB ;          DURCH USR("HHMMSST")
C095          1BEE ;
C095 20 32 B7 1C0F STRING       JSR LEN1            ;Y=STRINGLAENGE
C098 C0 07    1C2C              CPY #$07            ;STRING=7ZEICHEN?
C09A D0 40    1C4A              BNE ALSET           ;NEIN DANN ALARM
C09C          1C69 ;                 ZEIT STELLEN?
C09C          1C6C ;
C09C AD 0F DD 1C88              LDA CRB2            ;TIMER B IN CIA2
C09F 29 7F    1CA3              AND #$7F            ;BIT7 LOESCHEN!
C0A1 8D 0F DD 1CBF              STA CRB2            ;NORMALE UHRZEIT
C0A4          1CE4 ;                 IN ECHTZEITUHR CIA2
C0A4          1CE7 ;
C0A4          1D05 ;  AUSLESEN DES ZEIT-STRINGS
C0A4          1D08 ;
C0A4 A0 00    1D22              LDY #$00            ;ZAEHLER AUF 0
C0A6 A9 24    1D47 STELLEN      LDA #$24            ;BCD 24 STD-VERGL.
C0A8 20 FE C0 1D65              JSR ASCBCD          ;ZEICHENTEST UND
C0AB          1D8D ;                 UMWANDLUNG IN BCD-ZAHL
C0AB D0 02    1DAB              BNE STD12           ;STUNDEN UNGLEICH
C0AD          1DCF ;                 NULL ? DANN SPRUNG
C0AD A9 24    1DE6              LDA #$24            ;SONST = 24
C0AF          1DE9 ;
C0AF C9 13    1E08 STD12        CMP #$13            ;STUNDEN GROESSER
C0B1 90 07    1E2A              BCC STDSET          ;ODER GLEICH 12 ?
C0B3          1E4D ;                 NEIN, DANN SPRUNG
C0B3 F8       1E67              SED                 ;SONST DAVON BCD 12
C0B4 38       1E7B              SEC                 ;SUBTRAHIEREN
C0B5 E9 12    1E8B              SBC #$12            ;UND
C0B7 D8       1E91              CLD
C0B8 09 80    1EA9              ORA #$80            ;BIT7 SETZEN
C0BA          1EAC ;
C0BA 8D 0B DD 1ECF STDSET       STA TODHR           ;BCD-STUNDEN UND
C0BD          1EF7 ;                 AM/PM-FLAG IN TOD-CIA2
C0BD          1EFA ;
C0BD 20 FC C0 1F18              JSR ASCBCD1         ;ZEICHENTEST U.
C0C0          1F3F ;                 UMWANDELN IN BCD-ZAHL
C0C0 8D 0A DD 1F5A              STA TODMIN2         ;ERGEBNIS IN
C0C3          1F7F ;                 TOD-MINUTENREGISTER
C0C3          1F82 ;
C0C3 20 FC C0 1F9F              JSR ASCBCD1         ;DASSELBE FUER
C0C6 8D 09 DD 1FBB              STA TODSEC2         ;DIE SEKUNDEN
C0C9          1FBE ;
C0C9 20 66 C1 1FDA              JSR TEST            ;PRUEFEN,OB 1/10
C0CC          1FF9 ;                 SEKUNDEN=ZAHL
C0CC 8D 08 DD 2017              STA TOD10TH2        ;UND EINTRAGEN
C0CF          2039 ;                 INS TOD-REGISTER
C0CF          205D ;  DIE UHR BEGINNT JETZT ZU LAUFEN
C0CF          2060 ;
C0CF A9 00    207D              LDA #$00            ;KENNUNG FUER OK.
C0D1 4C 3C BC 20A2 AKKUFAC      JMP ACTOFC          ;AKKU ZUR UEBER-
C0D4          20C9 ;                 GABE INS BASIC IN FAC
C0D4          20CC ;
C0D4          20F5 ;********** FEHLER AUFGETRETEN ********
C0D4          20F8 ;
C0D4 68       2116 FEHLER       PLA                 ;JSR-ADRESSEN VOM
C0D5 68       212A              PLA                 ;STAPEL HOLEN
C0D6          212D ;
C0D6 68       2138 ERROR        PLA
C0D7 68       213E              PLA
C0D8          2141 ;
C0D8 A9 FF    2164 ERROR1       LDA #$FF            ;FEHLERKENNUNG IN
C0DA D0 F5    2180              BNE AKKUFAC         ;AKKU UND FAC
C0DC          2183 ;
C0DC          21AC ;ENDE DIESES TEILS D. UNBEDINGTEN SPRG.
C0DC          21D5 ;-------------------------------------
C0DC          21D8 ;
C0DC          2201 ;**** ALARMZEIT EINLESEN **************
C0DC          2204 ;
C0DC          222A ;  AUFRUF DURCH Z.B. USR("AHHMMSST")
C0DC          222D ;
C0DC C0 08    224B ALSET        CPY #$08            ; 8 ZEICHEN ?
C0DE D0 F8    2265              BNE ERROR1          ;NEIN=FEHLER
C0E0          2268 ;
C0E0 AD 0F DD 2273              LDA CRB2
C0E3 09 80    228E              ORA #%10000000      ;ALARMBIT
C0E5 8D 0F DD 22A1              STA CRB2            ;SETZEN
C0E8          22A4 ;
C0E8 A9 84    22C0              LDA #%10000100      ;ALARM-NMI
C0EA 8D 0D DD 22D5              STA ICR2            ;ZULASSEN
C0ED          22D8 ;
C0ED A9 3C    22F3              LDA #$3C            ;VERZOEGERUNGS-
C0EF 85 04    230D              STA VORW            ;WERT VORGEBEN
C0F1 85 02    2318              STA VERZ
C0F3 A9 FF    2336              LDA #$FF            ;EOR-WERT VORGEBEN
C0F5 85 03    2341              STA FARB
C0F7 A0 01    235F              LDY #$01            ;BUCHSTABE UEBERL.
C0F9 4C A6 C0 236D              JMP STELLEN
C0FC          2370 ;
C0FC          2373 ;
C0FC          239C ;*************************************
C0FC          23C5 ;UNTERPROGRAMM ZUR UMWANDLUNG DER ASCII
C0FC          23EC ;CODES IN BCD-ZAHLEN UND PRUEFUNG DER
C0FC          23FF ;EINGABE-ZEICHEN.
C0FC          2402 ;
C0FC A9 60    2427 ASCBCD1      LDA #$60            ;BCD 60 ALS GRENZE
C0FE          244F ;                 FUER MIN UND SEK WERTE
C0FE          2452 ;
C0FE 85 24    2465 ASCBCD       STA INDEX3
C100 20 13 C1 2482              JSR TEST1           ;PRUEFEN OB ZAHL
C103 0A       249A              ASL                 ;AUS LSB INS MSB
C104 0A       24AB              ASL                 ;SCHIEBEN
C105 0A       24B2              ASL
C106 0A       24B9              ASL
C107 85 25    24D7              STA INDEX4          ;UND ZW.SPEICHER
C109          24DA ;
C109 20 13 C1 24F7              JSR TEST1           ;NAECHSTE ZIFFER
C10C          2510 ;                 PRUEFEN
C10C 05 25    252C              ORA INDEX4          ;MSB AUS ZWSP.
C10E          2552 ;                 UND LSB ZUSAMMENOREN
C10E C5 24    256F              CMP INDEX3          ;UNTER GRENZW.?
C110 B0 C4    258D              BCS ERROR           ;NEIN=FEHLERAUSG.
C112          2590 ;
C112 60       2596              RTS
C113          2599 ;
C113          25C2 ;** PRUEFUNG OB ASCII-ZAHL VORLIEGT ***
C113          25C5 ;
C113 B1 22    25E8 TEST1        LDA (INDEX),Y       ;ZEICHEN EIN-
C115          2607 ;                 LESEN IN AKKU
C115 38       260D              SEC
C116 E9 30    2625              SBC #$30            ;< ASCII 0 ?
C118 90 BA    263D              BCC FEHLER          ;JA=FEHLER
C11A          2640 ;
C11A C9 0A    2659              CMP #$0A            ;>= ASCII : ?
C11C B0 B6    2671              BCS FEHLER          ;JA=FEHLER
C11E          2674 ;
C11E C8       2690              INY                 ;SCHLEIFENZAEHLER + 1
C11F 60       2696              RTS
C120          2699 ;
C120          26C2 ;**** ENDE PROGRAMMTEIL UHR STELLEN ***
C120          26C5 ;
C120          26EE ;-------------------------------------
C120          26F1 ;
C120          271A ;********** UHR LESEN ****************
C120          271D ;
C120          273E ;     GESCHIEHT DURCH USR(ZAHL)
C120          2741 ;
C120 A9 07    2761 ZAHLVAR      LDA #$07            ;STRINGLAENGE
C122 20 7D B4 277F              JSR STRINI8         ;SCHAFFT 7 BYTE
C125          27A7 ;                 PLATZ FUER STRING UND
C125          27CF ;                 LEGT START NACH $62/63
C125          27F6 ;                 SOWIE LAENGE NACH $61
C125          2814 ;                 (FAC $61-66)
C125          2817 ;
C125 A0 00    2831              LDY #$00            ;ZAEHLER AUF 0
C127 AD 0B DD 284F              LDA TODHR           ;STUNDE AUSLESEN
C12A          2878 ;                 DABEI WIRD GESAMTE ZEIT
C12A          28A1 ;                 ZWISCHENGESPEICHERT UND
C12A          28C7 ;                 ERST NACH LESEN DER
C12A          28EF ;                 1/10-SEK ZURUECKGEHOLT
C12A          2914 ;                 MIT AKTUELLEN WERT.
```

```
C12A            2917 ;
C12A 08         2935            PHP                     ;STATUS ZWISCHENSPEICH.
C12B            2938 ;
C12B 29 1F      294F            AND #$1F                ;=0001 1111
C12D            2978 ;              LOESCHEN DER AM/PM-FLAG
C12D C9 12      2996            CMP #$12                ;=0001 0010 =BCD12
C12F D0 02      29B0            BNE NO12                ;<>DANN SPRUNG
C131 A9 00      29CE            LDA #$00                ;SONST STATTDESSEN
C133            29D1 ;
C133 28         29F0 NO12       PLP                     ;STATUS ZURUECKHOLEN
C134 10 05      2A0A            BPL AM                  ;FALLS KEINE AM/
C136            2A2F ;              PM-FLAG GESETZT WAR
C136            2A32 ;
C136 F8         2A4C            SED                     ;SONST ADDIEREN VON
C137 18         2A62            CLC                     ;BCD 12 WEIL PM
C138 69 12      2A6D            ADC #$12
C13A D8         2A73            CLD
C13B            2A76 ;
C13B 20 55 C1   2A96 AM         JSR BCDASC              ;UP ZUR UMRECHNG
C13E            2ABC ;              VOM BCD IN ASCII UND
C13E            2AE5 ;              ABLEGEN IM STRING. HIER
C13E            2B02 ;              STUNDENWERT
C13E            2B05 ;
C13E AD 0A DD   2B22            LDA TODMIN2             ;DASSELBE FUER
C141 20 55 C1   2B3C            JSR BCDASC              ;MINUTENWERT
C144            2B3F ;
C144 AD 09 DD   2B5C            LDA TODSEC2             ;UND SEKUNDEN-
C147 20 55 C1   2B6F            JSR BCDASC              ;WERT
C14A            2B72 ;
C14A AD 08 DD   2B8C            LDA TOD10TH2            ;UND 1/10-
C14D 20 60 C1   2BA8            JSR BCDASC1             ;SEKUNDENWERT
C150            2BAB ;
C150 68         2BC7            PLA                     ;USR-STRING-ARGUMENT-
C151 68         2BE5            PLA                     ;RUECKSPRUNG VORBEREIT.
C152            2BE8 ;
C152 4C CA B4   2C06            JMP STRLIT67            ;BRINGT STRING
C155            2C29 ;              DESCRIPTOR IN DIE
C155            2C52 ;              DESCRIPTORTABELLE ($19-
C155            2C79 ;              $21),SETZT POINTER IN
C155            2CA2 ;              FAC(HIER $64/65) DARAUF
C155            2CC8 ;              SETZT STRING-FLAGGE,
C155            2CE9 ;              ERHOEHT LETZTEN
C155            2D10 ;              DESCRIPTOR-INDEX UM 3
C155            2D38 ;              ROUTINE ENDET MIT RTS.
C155            2D3B ;
C155            2D64 ;**** ENDE DES LESENS DER UHR *********
C155            2D67 ;
C155            2D90 ;---------------------------------------
C155            2D93 ;
C155            2DBC ;* UNTERPROGRAMM Z. UMRECHNUNG BCD IN *
C155            2DE5 ;* ASCII U.EINTRAGEN IN STRINGSPEICHER*
C155            2DE8 ;
C155 48         2E08 BCDASC     PHA                     ;AUF STAPEL ZW.SPEICH.
C156            2E0E ;
C156 4A         2E2A            LSR                     ;MSB INS LSB SCHIEBEN
C157 4A         2E31            LSR
C158 4A         2E38            LSR
C159 4A         2E3F            LSR
C15A            2E42 ;
C15A 20 60 C1   2E5E            JSR BCDASC1             ;IN ASCII UM-
C15D            2E85 ;              RECHNEN UND SPEICHERN
C15D            2E88 ;
C15D 68         2EA5            PLA                     ;ZURUECKHOLEN DER BCD-
C15E 29 0F      2EC3            AND #$0F                ;ZAHL,LOESCHEN DES
C160            2ED8 ;              MSB
C160            2EDB ;
C160 09 30      2F00 BCDASC1    ORA #$30                ;DAZUODERN VON $30
C162            2F28 ;              ERZEUGT (WEIL NUR ZAHL
C162            2F4F ;              ZWISCHEN 0 UND 9) DEN
C162            2F78 ;              ASCII-WERT ($30 BIS 39)
C162            2F7B ;
C162 91 62      2F98            STA (FAC1),Y            ;EINTRAGEN IN
C164            2FB7 ;              STRINGTABELLE
C164            2FBA ;
C164 C8         2FCC            INY                     ;ZAEHLER +1
C165            2FCF ;
C165 60         2FD5            RTS
C166            2FD8 ;
C166            3001 ;****** REST DES UP ASCII-BCD *********
C166            3004 ;
C166 20 13 C1   3020 TEST       JSR TEST1               ;PRUEFT AUF
C169            3042 ;              ASCII-ZAHL (0-9)
C169 60         3048            RTS
C16A            304B ;
C16A            3074 ;****** NMI-REAKTION AUF ALARM ********
C16A            3077 ;
C16A A9 77      3097 ALARM      LDA #L,ALIRQ            ;NEUER IRQ-
C16C 8D 14 03   30AB            STA IRQVL               ;VEKTOR
C16F A9 C1      30BA            LDA #H,ALIRQ
C171 8D 15 03   30C6            STA IRQVH
C174            30C9 ;
C174 4C BC FE   30E5            JMP NMIEND              ;REST DER NOR-
C177            3108 ;              MALEN NMI-ROUTINE
C177            310B ;
C177            3134 ;****** DIE  NEUE IRQ-ROUTINE *********
C177            3151 ;          RAHMENBLINKEN
C177            3154 ;
C177            3157 ;
C177 C6 02      3175 ALIRQ      DEC VERZ                ;ZEITSCHLEIFE
C179 F0 03      3191            BEQ BLINK               ;BLINKEN WENN 0
C17B            3194 ;
C17B 4C 31 EA   31B2            JMP NORM                ;SONST NORMALE IRQ
C17E            31B5 ;
C17E A5 04      31D5 BLINK      LDA VORW                ;ZAEHLER RUECK-
C180 85 02      31E8            STA VERZ                ;SETZEN
C182            31EB ;
C182 AD 20 D0   3203            LDA RAND                ;RAHMENFARBE
C185 45 03      321B            EOR FARB                ;INVERTIEREN
C187 8D 20 D0   3226            STA RAND
C18A            3229 ;
C18A 4C 31 EA   3244            JMP NORM                ;ZUM NORMAL-IRQ
C18D            3247 ;
C18D            324D            .EN
```

Listing 11. Der Quelltext zur Echtzeituhr (Schluß)