SYBEX

COMMODORE 128

Assembler-Kurs

Frank U. Müller

mit Programm-Diskette

Commodore 128
Assembler-Kurs

Commodore 128 Assembler-Kurs

Frank U. Müller

SYBEX

DÜSSELDORF · SAN FRANCISCO · PARIS · LONDON

Satz: SYBEX-Verlag GmbH, Düsseldorf
Umschlaggestaltung: tgr, unter Verwendung eines Commodore Pressefotos
Gesamtherstellung: Hub. Hoch, Düsseldorf

ISBN 3-88745-522-3
1. Auflage 1987

Inhaltsverzeichnis

Vorwort

Das vorliegende Buch besteht aus zwei Teilen, einem Assemblerkurs und dem Handbuch für den beiliegenden Assembler EDASS. Der Assemblerkurs führt Schritt für Schritt in die Programmierung des 6502- bzw. 8502-Prozessors ein. Jeder Befehl wird ausführlich erklärt und an Beispielen praktisch vorgeführt.

Diese Beipiele können mit dem Assembler EDASS nachvollzogen werden, denn der Kurs führt auch in die grundlegende Bedienung des Assemblers ein. An Hand von kleinen Aufgaben kann der Leser seinen Wissensstand testen und prüfen, ob er das Erlernte auch selbst in einem Programm einsetzen kann. Musterlösungen sind im Anhang A abgedruckt.

Neben diesen Kenntnissen der Assemblersprache führt der Kurs auch in die Bedienung eines Maschinensprache-Monitors ein und gibt Tips bei der Fehlersuche in einem Programm. Am Ende verfügt der Leser über genügend Kenntnisse, eigene Programme zu entwickeln.

Im zweiten Teil, dem Handbuch für den Assembler EDASS, werden dessen Befehle ausführlich erklärt. Der Leser wird in die Details des Assemblers eingeführt. Der erfahrene Assemblerprogrammierer, der vielleicht den Kurs übersprungen hat, findet hier alle nötigen Informationen über die Bedienung des Assemblers.

Der Name EDASS entstammt den beiden Wörtern EDitor und ASSembler. Der Editor ist ganz auf die Assemblerprogrammierung zugeschnitten. Die Befehlszeilen werden bei der Eingabe überprüft, in gekürzter Form abgespeichert und schließlich wieder formatiert ausgegeben.

Der Assembler arbeitet Label-orientiert, kennt bedingte Assemblierung und Makro-Assemblierung. Der Programmtext kann beim Assemblieren aus dem Speicher oder direkt von Diskette gelesen werden. Viele Pseudobefehle unterstützen bei der Programmerstellung. Natürlich arbeiten beide Programmteile sowohl im 40- als auch im 80-Zeichenmodus.

Viel Spaß mit dem Assemblerkurs und beim Programmieren mit EDASS!

Kapitel 1

Assemblersprache, was ist das?

Im Inneren Ihres Commodore 128 befindet sich der Mikroprozessor 8502. Dieser ist eine leicht veränderte Version des legendären 6502, der in vielen anderen Computern eingesetzt wird. Auch der 6510, der im Commodore 64 verwendet wird, stammt von diesem ab. Alle drei Prozessoren sprechen die gleiche Sprache. Am Ende des Kurses sind Sie also auch in der Lage, z.B. Programme für den C64 oder den Apple IIe zu schreiben.

Die Sprache, die Mikroprozessoren verstehen, wird als Maschinensprache bezeichnet. Der Name deutet schon daraufhin, daß diese Sprache besonders leicht von einer Maschine verstanden wird, aber für Menschen sehr unhandlich ist.

Der Mikroprozessor ist mit dem Computerspeicher verbunden. Aus diesem holt er sich einen Befehl in der Form eines Zahlenwertes und legt am Ende dort wieder ein Ergebnis ab. Zahlen in unserem Sinn gibt es für den Prozessor natürlich nicht. Für diesen besteht der Speicherinhalt aus elektrischen Impulsen. Die Impulse lassen sich leicht als binäre Zahlen darstellen.

Ein Maschinenprogramm, das einen kleinen Kreis auf dem Bildschirm darstellt, sieht für den Mikroprozessor wie folgt aus:

```
10101001 01010001 10001101 00000000 00000100 10101001
00000111 10001101 00000000 11011000 01100000
```

Aus dieser Folge von Nullen und Einsen ist kein Sinn herauszulesen. Auch durch eine Umwandlung der Binärzahlen in Dezimalzahlen wird das Programm nicht verständlicher.

```
169 081 141 000 004 169 007 141 000 216 096
```

Mit dieser Darstellung der Befehle ein Programm zu schreiben, das z.B. eine Länge von 500 Zahlen hat, ist eine sehr schwierige Arbeit, bei noch längeren Programmen eine fast unmögliche Aufgabe.

Um die Programmierung zu vereinfachen, hat man den Befehlen Buchstabenkürzel zugeordnet. In der Fachsprache werden diese als Mnemoniks bezeich-

net. Durch eine Zahl nach dem Befehl wird mitgeteilt, wo die zu verarbeitenden Daten im Speicher stehen. Das Programm von oben würde jetzt wie folgt aussehen:

```
lda   #91
sta   1024
lda   #7
sta   55296
rts
```

Die Befehlswörter sind die Abkürzungen englischer Ausdrücke, die die Funktion des Befehls beschreiben. So steht LDA z.B. für **"LoaD Accumulator"** (lade den Akkumulator).

Diese Darstellungsart der Maschinensprache wird als Assemblersprache bezeichnet. Dementsprechend heißt ein einzelner Befehl dann Assemblerbefehl und ein ganzes Programm Assemblerprogramm.

Die Buchstabenkürzel der Befehle hat man sich schnell eingeprägt. Die Erstellung eines längeren Programms wird damit sehr leicht. Nur kann der Prozessor diese Form der Befehle nicht direkt verstehen. Dazu muß für jeden Befehl wieder der ursprüngliche Zahlenwert eingesetzt werden. Diese Übersetzung von der Assemblersprache in die Zahldarstellung der Maschinensprache wird als Assemblieren bezeichnet. Nach dem Assemblieren würde das Programm wie folgt aussehen:

```
169 081        lda #81
141 000 004    sta 1024
169 007        lda #7
141 000 216    sta 55296
096            rts
```

Rechts stehen die Assemblerbefehle und links die Zahlenwerte der Maschinenbefehle. Dieses Endprodukt des Assemblierens wird auch Objektcode genannt. Es ist das eigentliche Maschinenprogramm.

Den Umwandlungsprozeß, also die Assemblierung, von Hand vorzunehmen, ist eine langwierige und langweilige Aufgabe, und es würden sich Fehler einschleichen. Zwar ist diese Art zu programmieren immer noch leichter als direkt mit den Zahlenwerten der Maschinenbefehle zu hantieren, aber der wahre Programmiergenuß ist es noch nicht.

Um langwierige und langweilige Aufgaben zu lösen, werden Computer gebaut. So läßt man den Assembliervorgang einfach vom Computer ausfüh- ren. Das entsprechende Programm heißt Assembler.

Damit wäre die Kette wieder geschlossen. Das Assemblerprogramm wird eingegeben, vom Computer umgesetzt und ausgeführt. Mit den Umwandlungsprozessen hat der Programmierer nichts zu tun. Für ihn genügt die genaue Kenntnis der Assemblersprache.

Assemblerprogramm mit Assemblerbefehlen
(Quelltext, dargestellt durch Buchstaben)
↓
Assembler
↓
Maschinensprache mit Maschinenbefehlen
(Objektcode, dargestellt durch Zahlenwerte im Speicher)

Der Quelltext mit seinen Assemblerbefehlen wird vom Programmierer eingegeben. Diese Form des Programms ist für den Prozessor nicht verständlich. Erst der Assembler übersetzt den Text für den Prozessor und erzeugt das eigentliche Maschinenprogramm. Dieses wird jetzt vom Prozessor verstanden.

Für den Programmierer ist diese Programmform zum Bearbeiten aber wieder unhandlich und ungeeignet. Der Assembler stellt das Bindeglied zwischen beiden Programmformen dar.

Kapitel 2

Ein erstes Programm

Starten des Assemblers EDASS

Schalten Sie Ihren Commodore 128 ein, und vergewissern Sie sich, daß der 128er-Modus aktiv ist. Als Bildschirmdarstellung wählen Sie bitte 40 Zeichen. Der Assembler kann zwar in beiden Modi arbeiten, jedoch gestalten sich die ersten Programmbeispiele im 40-Zeichen-Modus einfacher. Sobald der erforderliche Wissensstand erreicht ist, werden die Beispiele so angelegt, daß diese in beiden Modi arbeiten. Legen Sie jetzt bitte die EDASS-Diskette in das Diskettenlaufwerk ein, und starten Sie den Assembler mit

```
RUN "EDASS 128"
```

Nach kurzer Zeit erscheint die Startmeldung

```
**** EDASS - EDITOR+ASSEMBLER ****
      (c) BY FRANK MUELLER
         C128 VERSION

READY.
```

Auch nachdem EDASS aktiv ist, können Sie wie gewohnt in BASIC programmieren. Für Programmtext stehen Ihnen 11 KBytes bzw. bei eingeschalteter Grafik 2 KBytes zur Verfügung. Für BASIC-Variablen sind 3 KBytes reserviert.

Die einzelnen Funktionen von EDASS sind als Befehlserweiterung in den C128 eingebunden. Durch ein Ausrufezeichen ("!") am Zeilenanfang signalisieren Sie, daß ein EDASS-Befehl aufgerufen wird. Hinter dem Ausrufezeichen steht das Befehlswort, gefolgt von einigen Parametern. Ein Assemblerprogramm wird z.B. wie folgt von Diskette eingelesen:

```
!LOAD"PROGRAMM"
```

Alle EDASS-Befehle dürfen nur im Direktmodus verwendet werden. In einem BASIC-Programm führen sie zu einer Fehlermeldung.

Der Editor

Das Programm EDASS beinhaltet zwei wesentliche Komponenten, den Editor zur Programmerstellung und den Assembler zum Übersetzen der Programme. Der Editor kann bis zu zehn Programme gleichzeitig im Speicher verwalten. Um ein neues Programm im Speicher anzulegen, geben Sie bitte ein:

```
!BEGIN"KREIS"
```

Sie haben jetzt dem Editor mitgeteilt, daß Ihr neues Programm "KREIS" heißt. Unter diesem Namen wird es zum Editieren, Löschen, Speichern oder Assemblieren aufgerufen. Die Eintragung im Speicher läßt sich mit dem Befehl

```
!DISPLAY
```

überprüfen. Auf dem Bildschirm erscheint die Liste aller im Speicher befindlichen Programmme, in diesem Fall nur der Name "KREIS". Zusätzlich erfahren Sie, wie viele Zeilen Ihre Programme enthalten. Um das Programm einzugeben, wird der Editor mit

```
!EDIT"KREIS"
```

aufgerufen. Befinden sich mehrere Programme im Speicher, wird mit dem Programmnamen das gewünschte Programm ausgewählt. Ihr Bilschirm sieht jetzt wie in Abbildung 2.1 aus.

```
 PROGRAMMNAME    KREIS        -E0013-A:0006-E:0147:


START     LDA   #0                          ;RAHMENFARBE
          STA   53280
          LDA  #11                         ;HINTERGRUNDFARBE

          STA   53281
          RTS
>ENDE     .END                             ;*** ENDE ***
```

Abb. 2.1: Bildschirmdarstellung des Programms "Kreis"

In der obersten Bildschirmzeile erscheint die Statuszeile. Links lesen Sie den Namen des Programms, das gerade bearbeitet wird. In diesem Fall den Namen

"KREIS". Als nächstes folgt "E" für den Eingabemodus "Einfügen" und die aktuelle Eingabezeile. Die weiteren Informationen können vorerst unbeachtet bleiben. Die Statuszeile wird auch zur Ausgabe von Fehlermeldungen des Editors und des Assemblers verwendet. Die ursprünglichen Informationen verschwinden dabei für kurze Zeit.

In der Mitte des Bildschirms befindet sich deutlich markiert die Eingabezeile. Dort steht auch der Cursor und wartet auf Ihre Eingaben. An dieser markierten Stelle tippen Sie die Programmzeilen ein. Auch können Sie hier Befehle an EDASS senden. Es muß einfach am Zeilenanfang ein Ausrufezeichen stehen, gefolgt vom Befehl und den Parametern. Auf diese Weise können Sie z.B. aus dem Editor heraus ein neues Programm anfangen oder ein anderes von Diskette laden.

Das kurze Beispielprogramm aus Kapitel 1 soll nun eingegeben werden. Neben den eigentlichen Befehlen müssen dem Assembler noch verschiedene Anweisungen erteilt werden. Das vollständige Programm sieht dann wie folgt aus:

```
*=      $1300
.OBJ    M
.BANK   15
LDA     #81
STA     1024
LDA     #7
STA     55296
RTS
.END
```

Links steht der Befehl und rechts der dazugehörige Parameter. Beide Teile müssen durch eine Leerstelle voneinander getrennt werden. Tippen Sie jetzt die Programmzeilen ein. Die Eingabe funktioniert wie beim normalen Bildschirmeditor von BASIC. Der Cursor läßt sich in der Zeile frei bewegen, Zeichen können mit der <DEL>-Taste gelöscht und mit der <INS>-Taste eingefügt werden. Mit der <CLR>-Taste wird die Eingabezeile gelöscht, und mit <HOME> bewegt sich der Cursor an den linken Rand. Auch die meisten ESC-Sequenzen des normalen Editors stehen Ihnen zur Verfügung, genauso wie die Funktionstastenbelegung. Die Eingabezeile kann jedoch nicht verlassen werden.

Am Zeilenende drücken Sie jeweils die <RETURN>-Taste. Der Editor analysiert jede Zeile und gibt den Befehl und den Parameter wieder über der Eingabezeile aus. Mit jeder neuen Zeile rutschen die zuvor eingegebenen Zeilen etwas höher. Am Ende haben Sie das komplette Programm sauber formatiert

vor sich auf dem Bildschirm. Egal wie viele Leerstellen Sie zwischen Befehl und Parameter eingegeben haben, alle Befehle und alle Parameter stehen sauber untereinander. Sollten Sie sich bei der Eingabe einer Zeile vertippt haben und ein sinnloser Zeileninhalt entstanden sein, weist der Editor Sie mit der Meldung

```
?MNEMONIK NICHT VORHANDEN
```

auf Ihren Fehler hin. Der fehlerhafte Inhalt der Eingabezeile wird nicht gelöscht, damit Sie die Möglichkeit haben, den Fehler schnell zu beseitigen.

Eigentlich könnte das Programm jetzt dem Assembler übergeben werden. Bestimmt interessiert es Sie aber, wie Zeilen in den Programmtext eingefügt, geändert oder gelöscht werden. Spielen Sie dazu ein bißchen mit der <Cursor-Auf/Ab>-Taste.

Sie merken, wie sich der Programmtext entsprechend auf und ab bewegt. Die Eingabezeile stellt sozusagen die Lücke im Programmtext dar, in die eine neue Zeile eingegeben werden kann. Probieren Sie es doch einmal aus.

Geben Sie irgendeine Zeile aus dem Programm ein zweites Mal ein. Nach dem Drücken der <RETURN>-Taste erscheint die neue Zeile im Programmtext. Um Zeilen zu ändern, muß der Eingabemodus gewechselt werden. Dies geschieht durch Drücken der Tasten <SHIFT> und <RETURN>. Das Ergebnis sehen Sie sofort in der Statuszeile. Anstatt eines "E" wird jetzt ein "A", für "ändern", als Eingabemodus angezeigt. Befand sich die Eingabezeile beim Umschalten inmitten des Programms, wird die Programmzeile unter der Eingabezeile angezeigt.

Drücken Sie jetzt die <Cursor-Auf/Ab>-Taste, wird immer eine neue Programmzeile in der Eingabezeile angezeigt. Diese angezeigte Zeile können Sie wie bei einer Neueingabe ändern. Verändern Sie auf diese Weise eine Zeile des Programms.

Achten Sie darauf, daß am Ende wieder etwas Sinnvolles, z.B. eine der oben abgedruckten Programmzeilen, in der Eingabezeile steht. Am Ende müssen Sie natürlich wieder die Taste <RETURN> drücken. Zum Löschen einer Programmzeile bleibt der Änderungsmodus aktiv. Die gewünschte Zeile wird in die Eingabezeile bewegt. Mit der <CLR>-Taste wird die Eingabezeile gelöscht und mit der <RETURN>-Taste die Zeile im Speicher entfernt.

Anstatt die <RETURN>-Taste zu drücken, können Sie natürlich eine komplett neue Zeile eingeben und diese dann als Ersatz für die alte Zeile speichern. Das Ändern und Löschen einer Zeile ist ein endgültiger Schritt und läßt sich nicht rückgängig machen.

Solange die <RETURN>-Taste jedoch nicht gedrückt wurde, kann der alte Inhalt der Zeile mit der Taste <RUN/STOP> wiederhergestellt werden. Nach diesen kleinen Experimenten stellen Sie bitte den alten Programmtext, wie er auch oben abgedruckt ist, wieder her. Ändern und löschen Sie, wenn nötig, die entsprechenden Zeilen.

Der Editor läßt sich dabei sehr gut ausprobieren. Der Editor wird wieder verlassen. Schalten Sie dazu mit <SHIFT> und <RETURN> auf den Einfügemodus (ein "E" muß in der Statuszeile erscheinen). Tippen Sie als Befehl an EDASS ein:

```
!EXIT
```

Der Bildschirm wird gelöscht, und die "READY."-Meldung erscheint. Da der EXIT-Befehl häufig benötigt wird, kann dieser durch ein "Kleiner-als-Zeichen (<)" oder einen Pfeil nach links ("← ") abgekürzt werden. Dieses Zeichen wird wie ein Befehl in die Eingabezeile getippt.

Der Assembler

Der Assembler wird mit dem Befehl

```
!ASSEMBLER"KREIS"
```

gestartet. In Windeseile ist das Programm umgesetzt. Als Schlußmeldung erscheint

```
F130B-ENDE DER ASSEMBLIERUNG !
```

Sollte statt dessen eine Fehlermeldung ausgegeben worden sein, rufen Sie den Editor nochmals mit

`!EDIT"KREIS"` auf.

auf.Überprüfen Sie, ob der sichtbare Programmtext dem oben vorgegebenen entspricht, und korrigieren Sie die Unterschiede. Das Programm liegt jetzt für den Prozessor aufbereitet vor. Gestartet wird es durch

```
!GO$1300
```

Links oben auf dem Bildschirm erscheint ein gelber Kreis. Sollte dies nicht der Fall sein, stellen Sie fest, ob der 40-Zeichenmodus aktiviert und der Programmtext richtig eingegeben ist. Die weiteren Ausgaben des GO-Befehls sollen zunächst unbeachtet bleiben.

Die ersten Befehle

Sie sind sicher schon gespannt zu erfahren, wie das Programm den Kreis auf dem Bildschirm ausgibt. Zunächst zu den eigentlichen Anweisungen für den Prozessor:

```
lda    #81
sta   1024
lda   #7
sta   55296
rts
```

Jeder Befehl gliedert sich in zwei Teile: das Befehlswort und einen weiteren Parameter. Das Befehlswort wird Operationcode oder kurz Opcode genannt. In ihm steckt die Information, welche Funktion der Befehl erfüllt.

Der nachfolgende Parameter heißt Operand und bestimmt, woher der Befehl die Daten für seine Operation erhält: aus dem Programm, dem Speicher oder einer anderen Stelle. Die verschiedenen Möglichkeiten werden Adressierungsarten genannt. In Kapitel 5 folgt deren genaue Beschreibung.

Intern besitzt der Prozessor einen sogenannten Akkumulator. Dieser ist mit der Anzeige eines Taschenrechners vergleichbar. Daten werden in den Akkumulator eingelesen, also auf die Anzeige eingetippt, logisch mit anderen verknüpft, addiert oder subtrahiert. Am Ende kann der Inhalt des Akkumulators in den Speicher geschrieben werden, ähnlich dem Zwischenspeichern von Werten beim Taschenrechner.

Aber auch so simple Dinge wie den Inhalt einer Speicherzelle in eine andere zu übertragen lassen sich mit dem Akkumulator abwickeln. Der Akkumulator verarbeitet Zahlen von 0 bis 255.

LDA – LoaD Accumulator
– lade den Akkumulator

Der LDA-Befehl transportiert einen Wert in den Akkumulator. Das im Programm verwendete Zeichen "#" signalisiert dem Assembler, daß die nachfolgende Zahl unmittelbar in den Akkumulator geschrieben werden soll. Er erzeugt dann den entsprechenden Maschinenbefehl.

STA – STore Accumulator
– speichere den Akkumulator

Der STA-Befehl bewirkt genau das Gegenteil. Er transportiert den Inhalt des Akkumulators in den Speicher. Die dem Befehl folgende Zahl ist die Nummer bzw. Adresse der Speicherzelle. Der Inhalt des Akkumulators geht nicht verloren.

Das Speichern ist eher als Kopieren zu bezeichnen. Der Inhalt kann danach weiter verarbeitet oder nochmals im Speicher abgelegt werden.

Das Beispielprogramm lädt den Wert 81 in den Akkumulator und legt diesen in der Speicherzelle 1024 wieder ab. Diese Adresse entspricht der linken oberen Ecke des 40-Zeichen-Bildschirms. Dieser Speicherbereich umfaßt beim C128 25 mal 40, also 1000 Speicherstellen.

Belegt wird damit der Bereich 1024 bis 2023. Die ersten 40 Speicherzellen entsprechen der 1. Bildschirmzeile, die zweiten der 2. Zeile usw.. In einer Tabelle der Bildschirmcodes des C128, wie sie auch im Anhang H zu finden ist, können Sie nachlesen, daß 81 einem kleinen Kreis entspricht.

Es wird also ein kleiner Kreis in die linke obere Ecke geschrieben. Außer dem Zeichencode muß dem C128 der Farbwert des Zeichens mitgeteilt werden. Beim Löschen des Bildschirms wird der Farbspeicher mit der Hintergrundfarbe gefüllt.

Das neu geschriebene Zeichen wäre also nicht sichtbar. Ein graues Zeichen auf grauem Hintergrund kann man nicht sehen. Die Farbinformation des 40-Zeichen-Bildschirms liegt im Bereich 55296 bis 56295. Der Aufbau entspricht dem des Zeichenspeichers.

Wird in der Speicherstelle 55296, der linken oberen Ecke, z.B. der Wert 7 abgelegt, wird der Kreis gelb. Das Programm lädt dementsprechend eine 7 in den Akkumulator und speichert dessen Inhalt, also die 7, in der Speicherzelle 55296. Es erscheint ein gelber Kreis.

RTS – ReTurn from Subroutine
– kehre vom Unterprogramm zurück

Dieser Befehl ist mit dem BASIC-Befehl RETURN zu vergleichen. Der Prozessor kehrt zu dem Programm zurück, das das Beispielprogramm aufgerufen hat.

Bei EDASS ist dies das Programm des GO-Befehls. Am Ende eines Programms muß immer ein RTS-Befehl stehen, damit der Prozessor weiß, wo er seine Arbeit fortsetzen muß.

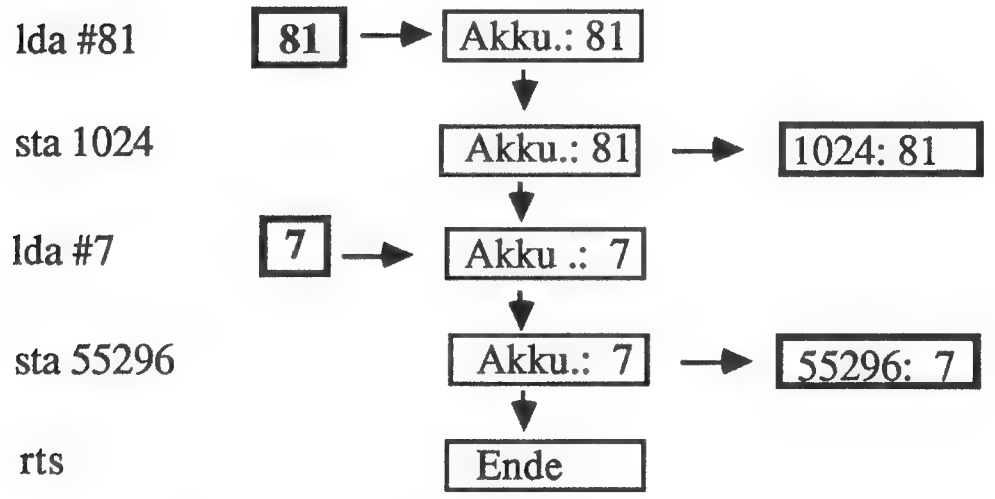


Abb. 2.2 Programmablauf

Den gesamten Programmablauf veranschaulicht nochmals Abb. 2.2. Der Akkumulator wird mit dem Code eines Kreises geladen und dieser in die Speicherstelle 1024 geschrieben. Dann wird der Akkumulator mit dem Code für gelbe Farbe geladen und im Farbspeicher bei der Adresse 55296 abgelegt. Am Ende wird das Programm wieder verlassen.

Anweisungen an den Assembler

Neben den Befehlen für den Prozessor enthält das Beispielprogramm Anweisungen für den Assembler. Diese Befehle werden, um sie von den Prozessoranweisungen zu unterscheiden, Pseudobefehle genannt. Bis auf wenige Ausnahmen werden diese durch einen Dezimalpunkt (".") eingeleitet.

```
*=        $1300
.OBJ       M
.BANK     15
 .         .
.
.END
```

Der .OBJ-Befehl legt fest, wohin das Ergebnis des Assemblierens, der Objektcode, geschrieben wird. "M" steht in diesem Fall für Memory, womit der Computerspeicher gemeint ist. Andere Angaben erlauben es, den Objektcode in einer Datei auf Diskette zu speichern oder auf dem Drucker auszugeben. Wird der .OBJ-Befehl ganz weggelassen, wird der Objektcode nirgendwo abgelegt. Der Assembler durchsucht den Programmtext nur nach Fehlern.

Der nächste Befehl .BANK steht im Zusammenhang mit der Speicheraufteilung des C128. In Kapitel 9 wird dies ausführlich beschrieben. Dort wird

auch der .BANK-Befehl genau erklärt. Der Assembler besitzt intern einen Zeiger, der bestimmt, an welche Stelle der nächste erzeugte Objektcode geschrieben wird.

Durch das Kommando *= wird diesem Zeiger ein Wert zugewiesen. Zu Beginn eines Programms wird damit die Adresse des ersten Befehls festgesetzt, also die Anfangsadresse des gesamten Programms. Mit dieser Startadresse wird das Programm auch später beim GO-Befehl aufgerufen. Natürlich kann der Zeiger auch mitten im Programm umgesetzt werden, womit ein Programm auf verschiedene Speicherbereiche verteilt wird.

Im Programmbeispiel wird die Adresse im Hexadezimalsystem angegeben. Eine dezimale Zahl an dieser Stelle ist natürlich genauso möglich. Um die Wirkung des Befehls zu testen, rufen Sie den Editor auf mit:

```
!EDIT"KREIS"
```

Ändern Sie die Zeile mit dem *=-Befehl in *= 5000 ab, verlassen Sie den Assembler wieder mit

```
!EXIT
```

und starten Sie den Assembler mit

```
!ASSEMBLER"KREIS"
```

Das Maschinenprogramm wurde jetzt an die Speicherstelle 5000 geschrieben. Entsprechend muß es jetzt auch mit dem Befehl

```
!GO5000
```

gestartet werden.

Der Parameter des GO-Befehl ist identisch mit der Startadresse des Maschinenprogramms im Computerspeicher. Der letzte Befehl im Programm ist die Anweisung .END. Diese teilt dem Assembler das Ende des Quellprogramms mit. Der Assembler erkennt zwar auch ohne .END-Befehl das Programmende, doch es gehört zu einem sauberen Programmierstil, ans Programmende auch einen .END-Befehl zu setzen.

Aufgabe 2.1 Schreiben Sie das Beispielprogramm so um, daß statt eines Kreises ein Stern erscheint. Versuchen Sie es gleich. Im Anhang H finden Sie eine Tabelle der Bildschirmcodes. Achten Sie beim Aufruf mit dem GO-Befehl auf die richtige Startadresse. Jeweils eine Musterlösung aller Aufgaben ist im Anhang A abgedruckt.

Sicher haben Sie sich beim Lösen der Aufgabe auch schon gedacht, daß der Name "KREIS" schlecht für ein Programmm geeignet, das einen Stern ausgibt. Dies läßt sich leicht ändern mit dem Befehl:

```
!RENAME"KREIS"="STERN"
```

Im RENAME-Befehl wird links der alte und rechts der neue Name eines Programms angegeben, beide werden durch ein Gleichheitszeichen ("=") getrennt. Unter dem neuen Namen "STERN" wird das Programm ab sofort aufgerufen.

Zum Schluß noch zwei weitere Befehle: Wenn Sie Ihr Programm auf Diskette sichern wollen, verwenden Sie den Befehl:

```
!SAVE"STERN"
```

Beim nochmaligen Speichern des Programms setzen Sie vor die Gänsefüßchen einen Klammeraffen ("@"):

```
!SAVE @"STERN"
```

Im deutschen Zeichensatz des C128 muß statt dessen das Paragraphenzeichen ("§") verwendet werden. Um das Programm später wieder in den Speicher einzulesen, dient der Befehl:

```
!LOAD"STERN"
```

Der Name wird automatisch in die Programmliste des Editors aufgenommen.

Ein weiteres Beispiel

Zum Schluß des Kapitels eine neue Variante des LDA-Befehls: Bei der oben kennengelernten Form mußte der Wert, der in den Akkumulator übertragen wird, direkt im Programm stehen. Das folgende Programm demonstriert, wie Werte auch aus einer Speicherzelle geladen werden können.

```
*=      $1300   ;STARTADRESSE FUER DAS PROGRAMM IST $1300
.OBJ    M       ;OBJEKTCODE IN DEN SPEICHER SCHREIBEN
.BANK   15      ;SPEICHERBANK 0 WAEHLEN
LDA     #1      ;DEN WERT 1 IN DEN AKKUMULATOR SCHREIBEN
STA     5000    ;DESSEN INHALT IN DIE SPEICHERZELLE 5000
LDA     5000    ;DEN INHALT DER ZELLE 5000 IN DEN AKKUMULATOR
STA     1024    ;DESSEN INHALT IN DIE SPEICHERZELLE 1024
STA     55296   ;UND NOCHMALS IN DIE SPEICHERZELLE 55296
RTS             ;DAS PROGRAMM WIEDER VERLASSEN
.END            ;ENDE DES PROGRAMMTEXTES
```

Wie Sie sehen, wurde das Programm jetzt mit einem Kommentar versehen. Dieser kann natürlich mit eingegeben werden. Der Assembler erkennt den Kommentar am vorangestellten Strichpunkt (";"). Der Rest der Zeile wird dann nicht mehr beachtet.

Es ist auch möglich, reine Kommentarzeilen einzugeben. Diese könnten z.B. für eine Kurzbeschreibung des Programms verwendet werden. Sie sollten sich angewöhnen, Ihre Programme immer zu kommentieren. Dies ist eine sehr brauchbare Hilfe, um auch nach Monaten noch zu verstehen, wie ein Programm eine bestimmte Aufgabe löst. Geben Sie das Programm jetzt ein. Hier nochmals die notwendigen Schritte:

1. neues Programm mit !BEGIN"BEISPIEL" anfangen
2. den Editor mit !EDIT"BEISPIEL" aufrufen
3. Zeile für Zeile eingeben
4. den Editor mit dem Befehl !EXIT wieder verlassen

Bei der Eingabe des Programms werden Sie eine Besonderheit des Editors bemerkt haben. Sobald der Cursor den rechten Rand des Bildschirms erreicht hat, rutscht der gesamte Inhalt (bis auf die Statuszeile) um ein Zeichen nach links.

Je mehr Zeichen Sie eingeben, desto weiter bewegt sich der Text nach links. Bewegen Sie den Cursor wieder zum linken Bildschirmrand, rutscht auch der Text in die entsprechend andere Richtung. Auf diese Weise können bis zu 80 Zeichen in die Eingabezeile getippt werden, obwohl der Bildschirm nur 40 Zeichen brei ist. Auf dem Bildschirm ist immer ein 40 Zeichen breites Fenster der 80 Zeichen zu sehen.

Man nennt deshalb einen derartigen Editor auch Pseudo-80-Zeichen-Editor. Leser, die nur den 40-Zeichen-Modus des C128 nutzen können, kommen so auch in den Genuß von 80-Zeichen-Eingabezeilen. Wenn Sie mit der Eingabe fertig sind, starten Sie den Assembler (!ASSEMBLER"BEISPIEL"). Sollte eine Fehlermeldung erscheinen, teilt der Assembler mit, in welchem Programm und in welcher Zeile der Fehler aufgetreten ist. Die Meldung selbst ist in deutsch abgefaßt und dürfte Ihnen genügend Aufschluß über die Art des Fehlers geben. Hier ein Beipiel für eine Meldung:

```
?UNBEKANNTE ADRESSIERART          0005/BEISPIEL
```

Sie teilt mit, daß ein Fehler im Operanden vorliegt. Der Fehler ist im Programm "BEISPIEL" in Zeile 5 aufgetreten. Wenn alles geklappt hat, kann das Programm gestartet werden mit:

```
!GO$1300
```

Als Ergebnis erscheint ein weißes "A" auf dem Bildschirm.

Jetzt zu der neuen Form des LDA-Befehls: Das Programm schreibt über den Akkumulator den Wert 1 in die Speicherzelle 5000. Der nächste LDA-Befehl liest den Inhalt dieser Speicherzelle wieder in den Akkumulator.

Die einfache Angabe einer Zahl hinter dem Mnemonik teilt dem Assembler mit, daß der Wert nicht direkt in den Akkumulator übernommen wird, sondern die Adresse einer Speicherzelle ist. Der Akkumulatorinhalt wird dann in die Speicherzelle 1024 kopiert, die linke obere Ecke des Bildschirms.

Der Inhalt des Akkumulators geht dabei nicht verloren und kann nochmals verwendet werden. In der nächsten Zeile wird er deshalb gleich in die Speicherzelle 55296 geschrieben, die linke obere Ecke des Farbspeichers.

Der Wert 1 bewirkt im Zeichenspeicher, daß ein "A" erscheint, und im Farbspeicher, daß dieses "A" weiß wird.

Aufgabe 2.2 Schreiben Sie ein Programm, das zwei rote Herzchen links oben auf den Bildschirm schreibt. Bei den Assembleranweisungen können Sie ja den bisherigen Programmkopf übernehmen. Für die Bildschirmzeichen und die Farbwerte finden Sie in Anhang H die nötigen Tabellen.

Zum Schluß noch ein neuer Befehl für den Editor. Um ein Programm im Speicher zu löschen und aus der Namensliste des Editors zu streichen, dient der Befehl !ERASE.

So löscht z.B. !ERASE "STERN" das Programm "STERN" im Speicher. Danach kennt der Editor das Programm "STERN" nicht mehr.

Kapitel 3

Daten transportieren und manipulieren

Addieren und subtrahieren

Die Beispiele im letzten Kapitel setzen den Akkumulator nur zum Transferieren von Daten ein. In diesem Abschnitt lernen Sie erste Befehle zur Manipulation des Akkumulatorinhalts.

ADC – ADd with Carry to accumulator
– addiere mit Übertrag zum Akkumulator

Dieser Befehl addiert den Akkumulatorinhalt zu dem Wert der Datenquelle, die im Operanden bestimmt wird. Das Ergebnis wird im Akkumulator gespeichert.

Was heißt nun "Wert aus einer Datenquelle"? Ähnlich wie beim LDA-Befehl kann mit dem Zeichen "#" ein Wert direkt aus dem Programm zum Akkumulator addiert werden. Der Wert kann aber auch aus einer Speicherzelle stammen. Sobald der Operand aus einer reinen Zahl besteht, wird diese Variante des Befehls vom Assembler gewählt. Als Beispiel soll auf dem Bildschirm erst eine weiße Null und dann eine gelbe Neun erscheinen.

```
*=$1300
.OBJ   M
.BANK  15
LDA    #48    ;AKKUMULATOR MIT DEM CODE EINER NULL LADEN
STA    1024   ;AKKUMULATORINHALT AUF DEN BILDSCHIRM
CLC           ;AKKUMULATOR FÜR DIE ADDITION VORBEREITEN
ADC    #9     ;PLUS 9 ERGIBT DEN CODE VON NEUN
STA    1025   ;AKKUMULATORINHALT AUF DEN BILDSCHIRM
LDA    #1     ;AKKUMULATOR MIT DEM CODE VON WEISS LADEN
STA    55296  ;AKKUMULATORINHALT IN DEN FARBSPEICHER
LDA    #6     ;AKKUMULATOR MIT DEM WERT 6 LADEN (DIFFERENZ)
CLC           ;DIE ADDITION VORBEREITEN
ADC    55296  ;PLUS DEN INHALT DER ZELLE 55296
STA    55297  ;AKKUMULATORINHALT IN DEN FARBSPEICHER
RTS           ;DAS PROGRAMM WIEDER VERLASSEN
.END
```

Geben Sie das Programm ein, und starten Sie es (mit !GO$1300). Auf dem Bildschirm erscheint, wie vorhergesagt, eine weiße Null und eine gelbe Neun. Das Programm verwendet beide besprochenen Formen des ADC-Befehls.

Zunächst wird, wie aus den vorangegangenen Beispielen bekannt, eine Null in den Bildschirmspeicher geschrieben. Anstatt danach den Akkumulator direkt mit dem Bildschirmcode der Neun zu laden, wird der Wert errechnet. Der CLC-Befehl bereitet den Akkumulator auf die Addition vor (die genaue Erklärung, warum dies gemacht werden muß, ergibt sich aus einem späteren Kapitel).

Der ADC-Befehl addiert den Wert 9 zum derzeitigen Inhalt des Akkumulators, hier 48. Das Ergebnis entspricht 57, dem Bildschirmcode einer Neun. Dieser kann sofort in den Bildschirmspeicher geschrieben werden. Um den Farbspeicher zu füllen, wird ein ähnlicher Weg beschrieben.

Zuerst wird die Farbe Weiß auf die bekannte Art in den Speicher transportiert. Für die zweite Farbe wird der Akkumulator direkt geladen und dann der Inhalt der Adresse 55296 addiert. Diese enthält eine 1, den Farbwert von Weiß. Als Ergebnis steht im Akkumulator 6+1, also 7.

Dieser Wert wird sofort in den Farbspeicher geschrieben. Die Vorgänge in diesem Programm sind schon komplizierter. Der Programmablauf und die Veränderung der Register- und Speicherinhalte sollen deshalb in einer Tabelle zusammengestellt werden.

Programm		AC	1024	1025	55296	55297
LDA	#48	48	???	???	???	???
STA	1024	48	48	???	???	???
CLC		48	48	???	???	???
ADC	#9	57	48	???	???	???
STA	1025	57	48	57	???	???
LDA	#1	1	48	57	???	???
STA	55296	1	48	57	1	???
LDA	#6	6	48	57	1	???
CLC		6	48	57	1	???
ADC	55296	7	48	57	1	???
STA	55297	7	48	57	1	7
RTS		7	48	57	1	7

In der Tabelle sind deutlich die Datentransfers und Veränderungen des Akkumulatorinhalts zu erkennen. Übrigens nennt man das handschriftliche Zusammenstellen einer derartigen Tabelle auch Schreibtischtest.

Kleinere Programme oder Abschnitte eines Programms werden auf dem Papier nachvollzogen. Ohne das Programm einzugeben, kann die Funktionsfähigkeit überprüft werden. Sollte bei einem eingegebenen Programm ein Fehler auftreten, kann der Schreibtischtest zur Fehlersuche benutzt werden. Alle Veränderungen von Registerinhalten können beobachtet werden.

Voraussetzung ist natürlich eine genaue Kenntnis der Wirkungsweise jedes Befehls. Im Beispielprogramm wurde jeweils mit dem kleineren Wert begonnen und der größere errechnet. Für den umgekehrten Weg ist ein Befehl zum Subtrahieren vom Akkumulatorinhalt notwendig.

SBC – SuBtract from accumulator with Carry
– subtrahiere vom Akkumulator mit Borgen

Wie beim LDA oder ADC-Befehl kann ein direkter Programmwert oder der Inhalt einer Speicherzelle vom Akkumulator subtrahiert werden. Das letzte Beispielprogramm, jetzt mit dem SBC-Befehl hat, folgendes Aussehen:

```
*=      $1300
.OBJ    M
.BANK   15
LDA     #57     ;AKKUMULATOR MIT DEM CODE EINER NEUN LADEN
STA     1025    ;AKKUMULATORINHALT AUF DEN BILDSCHIRM
SEC             ;AKKUMULATOR FUER DIE SUBTRAKTION VORBEREITEN
SBC     #9      ;MINUS 9 ERGIBT DEN CODE EINER NULL
STA     1024    ;AKKUMULATORINHALT AUF DEN BILDSCHIRM
LDA     #7      ;AKKUMULATOR MIT DEM CODE VON GELB LADEN
STA     55297   ;AKKUMULATORINHALT IN DEN FARBSPEICHER
LDA     #8      ;AKKUMULATOR MIT DEM WERT 8 LADEN (DIFFERENZ)
SEC             ;DIE SUBTRAKTION VORBEREITEN
SBC     55297   ;MINUS DEN INHALT DER ZELLE 55296
STA     55296   ;AKKUMULATORINHALT IN DEN FARBSPEICHER
RTS             ;DAS PROGRAMM WIEDER VERLASSEN
.END
```

Diesmal wird zuerst die Neun auf den Bildschirm geschrieben. Durch Subtraktion von 9 erhält man den Bildschirmcode einer Null. Auch hier muß der Akkumulator für die Subtraktion vorbereitet werden. Dies geschieht mit dem SEC-Befehl (die genaue Erklärung folgt später). Bei den Farbwerten wird auch zuerst die Farbe Gelb, in der die Neun erscheinen soll, in den Speicher geschrieben. Um die zweite Farbe zu erhalten, muß die Zahl 8 in den Akkumulator geschrieben werden. Durch Subtraktion des Farbwertes von Gelb, der in der Speicherzelle 55297 abgelegt ist, erhält man den Farbwert von Weiß, nämlich Eins.

Aufgabe 3.1 Geben Sie die Buchstaben A bis C in blauer Farbe auf dem Bildschirm aus. Verwenden Sie dabei den ADC- oder SBC-Befehl. Eine Musterlösung für beide Wege finden Sie wieder im Anhang A.

Das X- und Y-Register

Neben dem Akkumulator besitzt der Prozessor 8502 Prozessor zwei weitere Speicher zur Datenaufnahme, das X- und das Y-Register. Beide werden auch als Indexregister bezeichnet, da ihnen eine besondere Bedeutung bei der Adressierung von Speicherzellen zukommt. Beide können Werte zwischen 0 und 255 aufnehmen.

Verwendung finden sie hauptsächlich beim Datentransfer zwischen Speicherzellen, bei Zwischenspeichern von Werten und, durch ihre Funktion als Indexregister, beim Zugriff auf Tabellen. Wie beim Akkumulator können Werte in die Register geladen werden.

LDX – LoaD X-register
– lade das X-Register

Der entsprechende Befehl für das Y-Register lautet:

LDY – LoaD Y-register
– lade das Y-Register

Der übertragene Wert kann direkt aus dem Programm (z.B. LDX #12) oder aus dem Speicher stammen (z.B. LDY 1024).

STX – STore X-register
– speichere das X-Register

STY – STore Y-Register
– speichere das Y-Register

Diese beiden Befehle übertragen die Registerinhalte in den Speicher. An den letzten vier Befehlen erkennt man auch, daß die Mnemoniks (Befehlswörter) nicht willkürlich bestimmt wurden. Die Ladebefehle zum Einlesen eines Wertes in eines der drei Register des Prozessors fangen alle mit "LD" an. Der dritte Buchstabe bestimmt das Register.

Ein A steht für Akkumulator, ein X für das X-Register und ein Y für das Y-Register. Das gleiche gilt für die Speicherbefehle. Sie beginnen mit "ST" und der letzte Buchstabe wählt wieder das Register aus.

Das aus dem 1. Kapitel bekannte Beispielprogramm zur Darstellung eines Kreises kann auch ganz über das X-Register abgewickelt werden. Anstatt jeweils den Akkumulator zu laden und zu speichern, wird das X-Register verwendet.

```
*=      $1300
.OBJ    M
.BANK   15
LDX     #81     ;X-REGISTER MIT DEM WERT 81 LADEN
STX     1024    ;DESSEN INHALT IN DEN BILDSCHIRMSPEICHER
LDX     #7      ;X-REGISTER MIT DEM GELBEN FARBWERT LADEN
STX     55296   ;DESSEN INHALT IN DEN FARBSPEICHER
RTS             ;DAS PROGRAMM WIEDER VERLASSEN
.END
```

Das Programm liefert exakt das gleiche Ergebnis. Auch der Ablauf ist identisch, nur wird hier statt des Akkumulators das X-Register zum Datentransport eingesetzt.

Aufgabe 3.2 Stellen Sie mit einem Programm Ihren Vornamen auf dem Bildschirm dar. Verwenden Sie das X-Register bei der Übertragung der Zeichen und das Y-Register für die Farbe.

Von X nach Y über den Akkumulator

Mit den bisher gelernten Befehlen können auch Daten vom Akkumulator in eines der beiden weiteren Register oder zurück transportiert werden. Dies geht bislang nur mit dem Umweg über eine Speicherzelle. Folgender Programmausschnitt befördert den Inhalt des X-Registers in den Akkumulator:

```
.
.
STX 5000        ;X-REGISTER IN DIE SPEICHERZELLE 5000
LDA 5000        ;DEREN INHALT IN DEN AKKUMULATOR
.
.
```

Diese Methode ist sehr umständlich und benötigt viel Zeit. Im Befehlssatz befinden sich deshalb vier spezielle Befehle, die den Inhalt des Akkumulators in das X- bzw. Y-Register kopieren und zurück.

TAX – Transfer from Accumulator to X-register
– kopiere den Inhalt des Akkumulators ins X-Register

TAY – Transfer from Accumulator to Y-register
– kopiere den Inhalt des Akkumulators ins Y-Register

Den umgekehrten Weg erfüllen folgende Befehle:

TXA – Transfer from X-register to Accumulator
– kopiere den Inhalt des X-Registers in den Akkumulator

TYA – Transfer from Y-register to Accumulator
– kopiere den Inhalt des Y-Registers in den Akkumulator

Im Mikroprozessor sind die einzelnen Register miteinander verbunden. Für den Datenaustausch ist also kein Zugriff auf den Speicher notwendig. Der Befehl erfüllt seine Aufgabe dadurch sehr schnell. Im Programm ist auch keine Adresse im Operanden erforderlich, das Befehlswort allein genügt, um dem Prozessor mitzuteilen, woher die Daten stammen bzw. wohin sie fließen.

Zu den Befehlen ist noch zu sagen, daß der Inhalt der Register nicht vertauscht wird, sondern der Inhalt vom Quellregister ins Zielregister kopiert wird. Am Ende beinhalten beide den gleichen Zahlenwert. Auch für diese Befehle gibt es eine kleine Merkregel.

Alle vier Transferbefehle werden mit einem "T" eingeleitet. Der zweite Buchstabe ist das Quellregister, dessen Wert kopiert wird. Der dritte Buchstabe bestimmt das Zielregister. In dieses wird der Wert hineingeschrieben. Zwei weitere Transfer-Befehle, die später beschrieben werden, halten sich auch an diese Konvention.

Hier ein kleines Programm, das die Befehle einsetzt. Es wird ein grünes Dollarzeichen in die linke und rechte obere Ecke des Bildschirms geschrieben.

```
LDX    #36    ;X-REGISTER MIT CODE VON "$" LADEN
STX    1024   ;DAS ZEICHEN IN LINKE OBERE
STX    1063   ;UND IN DIE RECHTE UNTERE ECKE SCHREIBEN
TXA           ;DEN WERT 36 IN DEN AKKUMULATOR
SEC           ;EINE SUBTRAKTION VORBEREITEN
SBC    #31    ;MINUS 31 ERGIBT DEN FARBWERT VON GRUEN
TAY           ;DEN AKKUMULATORINHALT IN DAS Y-REGISTER
STY    55296  ;DEN FARBWERT IN DIE LINKE OBERE
STY    55335  ;UND DIE RECHTE OBERE ECKE SCHREIBEN
RTS           ;DAS PROGRAMM WIEDER VERLASSEN
```

Bei diesem Beispiel und bei allen zukünftigen werden die Assembleranweisungen weggelassen. Fügen Sie diese bei der Eingabe selbst hinzu, oder schreiben Sie sie von früheren Beispielen ab. Im Schnelldurchlauf nochmals die notwendigen Befehle.

```
*=     $1300  ;STARTADRESSE DES PROGRAMMS IST $1300
.OBJ   M      ;OBJEKTCODE IN DEN SPEICHER SCHREIBEN
.BANK  15     ;SPEICHERBANK 0 WAEHLEN
.
.
.END          ;DAS PROGRAMMENDE MARKIEREN
```

Gestartet wird das Programm wie üblich mit !GO$1300

Nun zur Erklärung des Programmablaufs. In den ersten drei Zeilen werden die Dollarzeichen über das X-Register in den Bildschirmspeicher geschrieben.

Der Farbwert wird dann durch eine Subtraktion berechnet. Da das X-Register diese nicht durchführen kann, wird mit dem TXA-Befehl der Wert in den Akkumulator kopiert und erst dort subtrahiert.

Als Ergebnis erhält man 5, den Farbwert von Grün. Dieser wird, als Abwechslung, jetzt ins Y-Register kopiert und aus diesem in den Farbspeicher geschrieben. Die Vorgänge dieses Programms sollen wieder in einer Tabelle dargestellt werden:

Programm		AC	XR	YR	1024	1063	55296	55335
LDX	#36	???	36	???	???	???	???	???
STX	1024	???	36	???	36	???	???	???
STX	1063	???	36	???	36	36	???	???
TXA		36	36	???	36	36	???	???
SEC		36	36	???	36	36	???	???
SBC	#31	5	36	???	36	36	???	???
TAY		5	36	5	36	36	???	???
STY	55296	5	36	5	36	36	5	???
STY	55335	5	36	5	36	36	5	5
RTS		5	36	5	36	36	5	5

Deutlich sind in der Tabelle die Datentransfers zwischen den Registern zu erkennen.

In diesem Beispielprogramm ist auch die Überschrift dieses Abschnittes verwirklicht. Ein Wert wird ins X-Register geladen, in den Akkumulator kopiert und von dort weiter in das Y-Register, also "von X nach Y über den Akkumulator".

Aufgabe 3.3 Schreiben Sie ein Programm, das einen weißen Dezimalpunkt in alle vier Ecken des Bildschirms setzt. Laden Sie dazu einen Wert direkt ins Y-Register, und verändern Sie diesen durch Addition oder Subtraktion (über Umwege ?!). Während dessen speichern Sie Zwischenwerte in den Bild- und Farbspeicher.

Plus eins und minus eins

Mit dem X- bzw. Y-Register können keine Werte addiert oder subtrahiert werden. Zwei Operationen erlauben es jedoch, den Inhalt zu verändern, nämlich um eins zu erhöhen bzw. um eins zu vermindern. Allgemein wird das Um-eins-Erhöhen auch, "Inkrementieren" und das Vermindern "Dekrementieren" genannt.

Die Befehle lauten für das X-Register wie folgt:

INX – INcrement X-register
– inkrementiere das X-Register

DEX – DEcrement X-register
– dekrementiere das X-Register

Für das Y-Register gilt entsprechend:

INY – INcrement Y-register
– inkrementiere das Y-Register

DEY – DEcrement Y-register
– dekrementiere das Y-Register

Wie bei den Transferbefehlen wird kein Zugriff auf den Speicher vorgenommen. Es ist deshalb keine Adresse im Operanden nötig. Alle vier Befehle sind zum Aufbau von Schleifen sehr wichtig. Meistens wird in diesen ein variabler Wert inkrementiert oder dekrementiert. Schleifen werden ausführlich in Kapitel 4 besprochen.

Auch die reine Eigenschaft, den Inhalt eines Registers zu verändern, wird sehr häufig ausgenutzt. Direkt aufeinanderfolgende Zahlenwerte können schnell in einem Register aufgebaut werden. Aus 2 wird durch Inkrementieren schnell 3 erzeugt und umgekehrt durch Dekrementieren wieder 2.

Das folgende Beispiel schreibt einen weißen Klammeraffen auf den Bildschirm. Der Klammeraffe hat den Bildschirmcode 0, und die Farbe Weiß hat den Wert 1. Beide Werte lassen sich also schnell durch einen Inkrementier-Befehl ineinander überführen. Realisiert mit dem Y-Register würde das Programm wie folgt ausssehen:

```
LDY     #0      ;WERT 0 IN DAS Y-REGISTER LADEN
STY     1024    ;DESSEN INHALT IN DEN BILDSCHIRMSPEICHER
INY             ;DURCH INKREMENTIEREN DEN WERT 1 ERHALTEN
STY     55296   ;1 ALS WEISSE FARBE IN DEN SPEICHER
RTS             ;DAS PROGRAMM WIEDER VERLASSEN
```

Noch ein Hinweis: In mancher Hinsicht sind das X-Register und das Y-Register mit dem Kilometerzähler eines Autos vergleichbar. Mit dem Inkrementier- Befehl lassen sich die Register wie Zähler verwenden. Ist der maximale Wert 255 erreicht, fangen die Register wie der Kilometerzähler wieder bei null an. Entsprechend wird bei Dekrementbefehlen verfahren. Wird der Wert Null dekrementiert, zeigt das Register als Ergebnis 255 an.

Aufgabe 3.4 Schreiben Sie das obige Beispielprogramm so um, daß statt des Y-Registers das X-Register und statt eines Inkrementier-Befehls ein Dekrementier-Befehl verwendet wird.

Aufgabe3.5 Versuchen Sie mit Hilfe der Inkrementier- oder Dekrementier-Befehle die Buchstaben A bis D in oranger Farbe links oben auf dem Bildschirm erscheinen zu lassen.

Binär- und Hexadezimalzahlen

Neben Dezimalzahlen werden im Assemblerprogramm häufig Binär- und Hexadezimalzahlen verwendet. Die gesamte Elektronik eines Computers kennt nur zwei Zustände, nämlich: es fließt kein Strom, oder es fließt Strom. Um diese Form der Daten auf dem Papier wiedergeben zu können, bedient man sich eines Zahlensystems, dessen Ziffern ebenfalls nur zwei Werte, nämlich 0 und 1, annehmen können.Eine einzelne Ziffer wird Bit und eine ganze Zahl Binärzahl genannt. Ein paar Beispiele für Binärzahlen sehen Sie hier:

```
10111010110
10101
101110101
101
```

Genau wie im Dezimalsystem haben auch hier die Ziffern einen Stellenwert. Im Dezimalsystem hat die niedrigste Stelle den Wert 1 (Einer), dann folgt 10 (Zehner), dann 100 (Hunderter) usw. Im Binärsystem hat die niedrigste Stelle ebenfalls den Wert 1. Es folgen 2, 4, 8 usw.

```
... 256 128 64  32  16  8   4   2   1
    0   1   0   1   0   1   1   1   0
```

Um den dezimalen Wert einer Binärzahl zu erhalten, müssen die einzelnen Stellen, multipliziert mit ihrem Stellenwert, addiert werden:

```
10101110  =  1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 1*2 + 0*1
          =  128 + 32 + 8 + 4 + 2
          =  178
```

Um die Länge von Binärzahlen zu standardisieren, hat man 4 Bits, also 4 Binärstellen, zu einem Nibble und 8 Bits, also zwei Nibbles bzw. 8 Binärstellen zu einem Byte zusammengefaßt. Wenn Ihr C128 also 128000 Bytes Speicherplatz besitzt, entspricht dies 1024000 Bits. Der Prozessor verarbeitet

die Bits nicht einzeln, sondern parallel. Der 8502 bearbeitet 8 Bits auf einmal, weshalb er auch zur Gruppe der 8-Bit-Prozessoren gehört. In größeren Computern rechnen die Prozessoren mit 16 oder gar 32 Bits. Die kleinste mit 8 Bits darstellbare Zahl ist 0 und die größte 255.

```
00000000  =  0
11111111  =  128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

Jetzt wird auch verständlich, warum Akkumulator, X- und Y-Register nur Zahlen in diesem Bereich speichern können. Auch jede Speicherzelle im Computer umfaßt 8 Bits und ist damit an diese Zahlgrenzen gebunden. Hier noch zwei weitere Konventionen, die Ihnen im Zusammenhang mit Bits und Bytes begegnen können. Die Bits werden von rechts nach links, beginnend bei 0, durchnumeriert. Zusätzlich wird das niederwertigste Bit LSB (least significant bit) und das höchstwertige Bit MSB (most siginificant bit) genannt.

```
Bit   7  6  5  4  3  2  1 0
      1  0  1  0  1  1  1 0
  MSB                    LSB
```

Binärzahlen mit ihren 8, 16 oder mehr Stellen sind in vielen Fällen unhandlich. Man führte deshalb ein Zahlensystem ein, bei dem jede Stelle 16 Werte annehmen darf, das Hexadezimalsystem. Da vom Dezimalsystem nur 10 Ziffern zur Verfügung stehen, werden für die weiteren Ziffern die Buchstaben A bis F verwendet.

dezimal	hexadezimal	binär
0	0	00000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1100
12	C	1101
13	D	1110
14	E	1101
15	F	1111

Mit einer einzigen Stelle kann im Hexadezimalsystem bis 15 gezählt werden. Wie zu erkennen ist, stehen Binärzahlen und Hexadezimalzahlen in einer direkten Beziehung. Vier Bits werden zu einer Hexadezimalstelle zusammengefaßt. Folglich bilden zwei Hexadezimalziffern ein Byte.

Vier Ziffern, also zwei Bytes, bilden eine Adresse. Zahlen dieser Größe werden Adresse genannt, da mit ihnen jede Speicherzelle, die der Prozessor erreicht, bezeichnet werden kann. Eine andere Bezeichnung für diese Zahlenlänge, nämlich "Wort", liest man vor allem im Zusammenhang mit 16- und 32-Bit-Prozessoren.

Zur Umrechnung von Hexadezimalzahlen in Dezimalzahlen wird die Summe der Stellen, multipliziert mit ihrem Stellenwert, gebildet. Die Stellen haben die Wert 1, 16, 256, 4096 usw.

B75F = 11*4096 + 7*256 + 5*16 + 15*1 = 46943

Binär- und Hexadezimalzahlen bei EDASS

Binärzahlen werden durch ein Prozentzeichen (%) und Hexadezimalzahlen durch ein Dollarzeichen ($" eingeleitet. Das Programm interpretiert die folgenden Zeichen entsprechend. Ein paar Beispiele für Zahlen:

%1010	Binärzahl
123	Dezimalzahl
$ABCD	Hexadezimalzahl
101	Dezimalzahl
$123	Hexadezimalzahl
%1011110	Binärzahl

Bei Befehlen dürfen diese Zahlensysteme beliebig gemischt werden. Es ist sogar möglich, im Operanden berechnete Ausdrücke zu verwenden. Hier einige Beispiele für den gleichen Befehl:

```
LDA   # 50
LDA   # $32
LDA   # 19 + $1F
LDA   # 5 * 10
LDA   # 100 / %10
```

Um das Ergebnis eines solchen Ausdrucks zu berechnen, kennt EDASS ein Kommando, das einen beliebigen mathematischen Ausdruck (nur EDASS-Rechenfunktionen) auswertet, und das Ergebnis auf dem Bildschirm darstellt.

`!=19+$1f` ergibt auf dem Bildschirm 50.

Das kann natürlich nicht nur dezimal erscheinen:

`!$=19+$1f` für hexadezimale Ausgabe und

`!%=19+$1f` für binäre Ausgabe.

Links vom =-Zeichen steht also das Zahlensystem für die Ausgabe und rechts ein beliebiger mathematischer Ausdruck (nur EDASS-Rechenfunktionen). Werden diese Kommandos im Editor aufgerufen, erscheint das Ergebnis in der Statuszeile.

In den bisherigen Beispielen wurde für die Anfangsadresse des Programms (*= $1300) das Hexadezimalsystem verwendet. Die Adresse $1300 ist der Start eines freien Speicherbereichs im C128. Im Dezimalsystem wäre dies $1300 = 4864. Diese Zahl ist wesentlich schlechter zu merken. Das Hexadezimalsystem hat also oft Vorteile gegenüber dem Dezimalsystem.

In Zukunft werden alle Hexadezimalzahlen, die im Text vorkommen, mit einem Dollarsymbol versehen. Eine Verwechslung mit Dezimalzahlen ist dann nicht mehr möglich.

Aufgabe 3.6 Experimentieren Sie ein bißchen mit den Zahlensystemen. Geben Sie sich selbst Zahlen vor, und wandeln Sie diese von Hand in ein anderes System um. Das Ergebnis kann mit dem =-Befehl überprüft werden.

Kapitel 4

Von Sprüngen zu Schleifen

Der Programmzähler

Die bisher behandelten Register waren für die Datenaufnahme bestimmt. Daneben gibt es auch solche, die der Programmabarbeitung dienen. Eines dieser Register ist der Programmzähler, manchmal auch als Programmzeiger bezeichnet. Wie der Name schon andeutet, beinhaltet dieses Register die Speicheradresse des gerade bearbeiteten Befehls. Wird gerade ein Befehl in der Speicherzelle 5000 bearbeitet, beinhaltet er den Wert 5000. Nach Beendigung des Befehls zeigt er auf das folgende Kommando, z.B. auf die Adresse 5002.

Sprünge

JMP – JuMP
– springe zu einer anderen Programmposition

Dieser Befehl lädt einen neuen Wert in den Programmzähler. Die Folge ist, daß der nächste Befehl nicht auf den gerade bearbeiteten folgt, sondern an der neuen durch den JMP-Befehl bestimmten Adresse. Der Programmzähler ist im Gegensatz zu allen anderen Registern 16 Bit lang. Dies entspricht den Zahlen von 0 bis 65535. Der maximale vom 8502 ansprechbare Speicher beträgt also 65536 Bytes oder kurz 64 KBytes (1 KByte = 1024 Bytes). Innerhalb dieses *Adreßraums*, wie der ansprechbare Bereich auch genannt wird, muß jedes Programm liegen. Auch alle Speicherzellen zur Aufnahme von Daten müssen sich dort befinden. Für den Prozessor existieren keine Speicherzellen mit einer Adresse größer als 65535. Mit welchen Umwegen es möglich ist, die 128 KByte des C128 anzusprechen, wird in einem eigenen Kapitel beschrieben.

Übrigens 65535 im Hexadezimalsystem lautet $FFFF, die ist die größte mit vier Stellen darstellbare Zahl. Im Binärsystem wären es entsprechend 16 Einsen. Diese Maximalgrenzen werden vom Assembler bei der Zahleingabe über- prüft. Ein Befehl wie LDA 100000 oder STX 654321 wird nicht assembliert, sondern ergibt eine Fehlermeldung. Doch wieder zurück zum JMP-Befehl. Im Operanden wird die neue Programmadresse angegeben. In einem Programm sähe dies wie folgt aus:

```
Adresse        Befehle
          .
            .
9000   JMP    10000
          .
          .
10000    LDA       ...
          .
          .
```

Bei Assemblerprogrammen wird nur die Startadresse vorgegeben. Die Adressen der einzelnen Befehle werden vom Assembler berechnet und bleiben Ihnen als Programmierer verborgen.

Folglich ist es unmöglich, bei JMP im Operanden die Adresse eines Befehls zu nennen, da diese nicht bekannt ist. Hier hilft eine Eigenschaft des Assemblers weiter, die Label-orientiertheit genannt wird. Jeder Programmzeile kann ein Name gegeben werden. Diese Kennzeichnung wird als Label oder Marke bezeichnet.

```
                .
                .
START      LDA     #10
           LDX     #23
SCHLEIFE   STX     6100
                .
                .
```

Beim Assemblieren werden die Namen der Labels zusammen mit der Adresse des Befehls gesondert gespeichert. Um jetzt die Adresse des Befehls zu erhalten, muß nur der Name des Labels im Speicher gesucht werden. Der beigefügte Wert ist die Adresse des Befehls. Ähnlich wie der Name einer BASIC-Variablen kann der Name des Labels an beliebiger Stelle in einem mathematischen Ausdruck erscheinen. Befehle könnten jetzt z.B. wie folgt aussehen:

```
              .
              .
STA      START+1
LDX      SCHLEIFE
JMP      SCHLEIFE
              .
              .
```

Am letzten Befehlsbeispiel wird klar, wie Sprunganweisungen mit einem Assembler am besten gelöst werden. Im Operanden steht einfach das Label des

Zielbefehls. Die Befehlsadresse wird automatisch eingesetzt. Damit hier kein falscher Eindruck entsteht: Sie können natürlich im Operanden auch Adressen in Zahlform verwenden.

Beim Aufruf von Routinen aus dem ROM des Computers ist dies sogar oft erforderlich. Im folgenden Beispielprogramm ist der Einsatz eines Sprungbefehls zwar sinnlos, es wird jedoch gut der Einsatz des Befehls demonstriert. Als Ergebnis erscheint auf dem Bildschirm ein graues C.

```
        LDA  #3      ;AKKUMULATOR MIT DEM WERT 3 LADEN
        STA  1024    ;DESSEN INHALT IN DIE LINKE OBERE ECKE
        JMP  FARBE   ;SPRINGEN ZU: FARBINFORMATION SCHREIBEN
ENDE    RTS          ;DAS PROGRAMM WIEDER VERLASSEN
FARBE   LDA  #15     ;AKKUMULATOR MIT FARBWERT GRAU LADEN
        STA  55296   ;DESSEN INHALT IN DEN FARBSPEICHER
        JMP  ENDE    ;SPRINGE ZU: DAS PROGRAMM BEENDEN
```

Zur Verdeutlichung soll wieder der Programmablauf aus der Sicht des Prozessors in einer Tabelle dargestellt werden.

Auch der Programmzähler PC (program counter) ist eingetragen.

Programm:		PC	AC	1024	55296
LDA	#3	4864	3	???	???
STA	1024	4866	3	3	???
JMP	4871	4867	3	3	???
	↓				
	Sprung				
	↓				
LDA	#15	4871	15	3	???
STA	55296	4873	15	3	15
JMP	4870	4876	15	3	15
	↓				
	Sprung				
	↓				
RTS		4870	15	3	15

Die Arbeit des Assemblers kann auch während des Assemblierens ausgegeben werden. Mit der Assembleranweisung ".LIST 3" erscheint auf dem Bildschirm das sogenannte Assemblerlisting.

Fügen Sie als erste Programmzeile diesen Befehl ein. Als Ergebnis erhalten Sie folgenden Text auf dem Bildschirm:

```
                        0001            *=      $1300
                        0002            .OBJ    M
                        0003            .BANK   15
F1300    A9 03          0004            LDA     #3        ;AKKUMULATOR MI...
F1302    8D 00 04       0005            STA     1024      ;DESSEN INHALT ...
F1305    4C 09 13       0006            JMP     FARBE     ;SPRINGEN ZU: F...
F1308    60             0007  ENDE      RTS               ;DAS PROGRAMM W...
F1309    A9 0F          0008  FARBE     LDA     #15       ;AKKUMULATOR MI...
F130B    8D 00 D8       0009            STA     55296     ;DESSEN INHALT ...
F130E    4C 08 13       0010            JMP     ENDE      ;SPRINGE ZU: DA...
                        0011            .END
F1311-ENDE DER ASSEMBLIERUNG!
```

In der ersten Spalte steht die Speicheradresse des Befehls, ausgedrückt in einer Hexadezimalzahl. Der zusätzliche Buchstabe "F" hängt mit der Speicherwahl des .BANK-Befehls zusammen. In den nächsten drei Spalten erscheint der Objektcode der Befehle.

Bei den beiden JMP-Befehlen sieht man deutlich, wie der Assembler für die Labels aus dem Programmtext die tatsächlichen Adressen der Befehle eingesetzt hat. In der nächsten Spalte steht die Zeilennummer der Programmzeile. Mit dieser Angabe kann man z.B. eine fehlerhafte Zeile zum Ändern wiederfinden. Schließlich wird rechts der eigentliche Text der Programmzeile ausgedruckt.

Bei Assembleranweisungen wie .BANK, .OBJ oder .END, die keinen Objektcode erzeugen, erscheinen die ersten vier Spalten nicht. Das zweite Beispielprogramm demonstriert die Möglichkeit, Programmteile in unterschiedlichen Speicherbereichen abzulegen. Beide Stücke werden durch einen JMP-Befehl beim Ablauf verbunden.

```
        *=      $1300   ;STARTADRESSE SETZEN
        LDA     #3      ;AKKUMULATOR MIT CODE VON "C" LADEN
        STA     1024    ;DIESEN WERT IN DEN BILDSCHIRMSPEICHER
        JMP     FARBE   ;SPRUNG ZUM 2. PROGRAMMTEIL
;
        *=      $1400   ;STARTADRESSE DES 2. TEILS SETZEN
FARBE   LDA     #12     ;AKKUMULATOR MIT DER FARBE GRAU LADEN
        STA     55296   ;DIESEN WERT IN DEN FARBSPEICHER
        RTS             ;DAS PROGRAMM WIEDER VERLASSEN
```

Das Beispiel zeigt auch, wie mit einer leeren Kommentarzeile der Programmtext gegliedert werden kann. Die beiden Programmteile sind bereits deutlich im Text zu erkennen.

Bei größeren Programmen können auf diese Weise Schleifen hervorgehoben, Unterprogramme abgetrennt und andere Strukturierungen angedeutet werden. Dies gehört, wie das Kommentieren des Programms, zu einem guten Programmierstil.

JSR – Jump to SubRoutine
– springe zu einem Unterprogramm

Wie JMP lädt auch dieser Befehl einen neuen Wert in den Programmzähler und springt damit an eine neue Programmposition. Zuvor wird jedoch der aktuelle Wert des Zählers zwischengespeichert (wo, spielt vorerst keine Rolle). Mit dem RTS-Befehl wird der zwischengespeicherte Wert wieder in den Programmzähler zurückgeladen.

Beide Kommandos werden für den Aufbau von Unterprogrammen verwendet. Ein solches wird mit JSR und nachfolgender Startadresse aufgerufen und wieder durch RTS verlassen, egal, von welcher Adresse der Aufruf erfolgte. Der Vorgang ist mit den beiden BASIC-Befehlen GOSUB und RETURN zu vergleichen. Auch Schachtelungen sind wie bei diesen erlaubt.

Unterprogrammstrukturen lassen sich nicht mit dem JMP-Befehl realisieren. Würde ein Unterprogramm von verschiedenen Stellen des Programms aufgerufen, könnte das Unterprogramm nicht entscheiden, zu welcher Adresse im Hauptprogramm es zurückkehren soll.

Es wird Ihnen jetzt sicher klar sein, warum am Ende jedes Beispielprogramms ein RTS-Kommando eingegeben werden muß. Der GO-Befehl ruft das Programm wie ein Unterprogramm auf. Die Startadresse des Unterprogramms, sozusagen der Operand des JSR-Befehl, wird von Ihnen beim GO-Befehl eingegeben.

Der RTS-Befehl verläßt das Programm und kehrt zur GO-Routine zurück. Das Beispielprogramm zum JMP-Befehl wird jetzt mit dem JSR-Befehl aufgebaut.

```
        LDA   #3      ;AKKUMULATOR MIT DEM WERT 3 LADEN
        STA   1024    ;DESSEN INHALT IN DIE LINKE OBERE ECKE
        JSR   FARBE   ;SPRINGEN ZU: FARBINFORMATION SCHREIBEN
        RTS           ;DAS PROGRAMM WIEDER VERLASSEN
;
FARBE   LDA   #15     ;AKKUMULATOR MIT FARBWERT GRAU LADEN
        STA   55296   ;DESSEN INHALT IN DEN FARBSPEICHER
        RTS           ;INS HAUPTPROGRAMM ZURÜCKKEHREN
```

Das Programm schreibt zuerst ein C in den Bildschirmspeicher und ruft dann das Programm zum Schreiben der Farbinformation als Unterprogramm auf.

Dabei merkt sich der Prozessor die Position im Hauptprogramm. Die Farbinformation wird geschrieben, und mit dem RTS-Befehl holt der Prozessor die gespeicherte Adresse wieder in den Programmzähler. Das Programm setzt

seine Arbeit wieder im Hauptprogramm fort und kehrt ebenfalls mit einem RTS-Befehl zur Routine des GO-Befehls zurück.

Aufgabe 4.1 Schreiben Sie das zweite Beispielprogramm zum JMP-Befehl so um, daß statt dessen der JSR-Befehl Verwendung findet.

Mit dem JSR-Befehl lassen sich jetzt auch Routinen aus dem ROM des Computers aufrufen, z.B. die BSOUT-Routine (BaSic OUTput) aus dem Betriebssystem des C128. Diese gibt ein Zeichen, das im ASCII-Code vorliegen muß, auf dem aktiven Ausgabegerät aus. Zur Übergabe wird der Akkumulator verwendet. Er muß den Code des Zeichens enthalten. Normalerweise erfolgt die Ausgabe auf dem Bildschirm, mit dem BASIC-Kommando CMD läßt sich aber auch ein anderes Gerät bestimmen.

Übrigens muß die Speicherverwaltung des C128 mit der Anweisung .BANK 15 auf den ROM-Zugriff vorbereitet werden.

```
LDA  #65      ;ASCII CODE VON "A" IN DEN AKKUMULATOR
JSR  $FFD2    ;DEN AKKUMULATORINHALT AUSGEBEN
LDA  #13      ;WAGENRÜCKLAUF IN DEN AKKUMULATOR
JSR  $FFD2    ;DEN AKKUMULATORINHALT AUSGEBEN
RTS           ;DAS PROGRAMM WIEDER VERLASSEN
```

Nach dem Start des Programms erscheint unter dem eingegebenen GO-Befehl ein A in der aktuellen Zeichenfarbe. Dieses kleine Beispielprogramm funktioniert auch auf dem 80-Zeichen-Bildschirm. Die Routine BSOUT spricht beide Darstellungen an. Sollten Sie über einen Monitor verfügen, probieren Sie es doch einfach einmal aus. Schalten Sie mit ESC+X auf 80-Zeichen-Ausgabe, und starten Sie das Programm nochmals.

Auch jetzt erscheint auf dem Bildschirm ein "A". Im 80-Zeichen-Modus darf auch der Editor aufgerufen werden. Die gesamte Zeile ist jetzt auf einen Blick zu überschauen. Der Text wird nicht, wie bei der 40-Zeichen-Darstellung, nach links und rechts verschoben. Zusätzlich wird in der Statuszeile der Eingabemodus in einem vollen Wort ausgeschrieben, z.B. "AENDERN" und nicht bloß "A".

Hier noch ein Tip: Schalten Sie Ihren C128 im 80-Zeichen-Modus mit dem BASIC-Befehl FAST auf die schnellere Arbeitsgeschwindigkeit. Die langsame Ansteuerung der 80-Zeichen-Darstellung wird damit wieder ausgemerzt. Das Beispiel hat noch einen kleinen Makel.

Beim Aufruf der Routine muß immer eine Zahl eingegeben werden. Dies ist eine große Fehlerquelle, Tippfehler werden sich immer wieder einschleichen. Hier helfen wieder Labels weiter. Diesen können auch direkt Werte zugewie-

sen werden. Außerdem wird, um die Lesbarkeit des Programms zu erhöhen, anstatt des ASCII-Codes, das Zeichen selbst im Programmtext angegeben. Erscheint eine Zeichenkette, eingeschlossen durch Gänsefüßchen, in einem mathematischen Ausdruck, nimmt EDASS den ASCII-Wert des ersten Zeichens für die weiteren Berechnungen.

```
BSOUT =     $FFD2   ;STARTADRESSE VON BSOUT DEFINIEREN
      LDA   #"A"    ;ASCII CODE VON "A" IN DEN AKKUMULATOR
      JSR   BSOUT   ;DEN AKKUMULATORINAHLT AUSGEBEN
      LDA   #13     ;WAGENRÜCKLAUF IN DEN AKKUMULATOR
      JSR   BSOUT   ;DEN AKKUMULATORINHALT AUSGEBEN
      RTS           ;DAS PROGRAMM WIEDER VERLASSEN
```

In der ersten Zeile wird mit dem =-Befehl das Label BSOUT definiert und der Wert $FFD2 zugewiesen. Wird die Routine später im Programm aufgerufen, kann statt eines Zahlenwertes im Operanden der Name des Labels stehen. Das Label muß nicht unbedingt BSOUT heißen, man sollte aber immer darauf achten, daß sinnvolle Namen gewählt werden, die z.B. schon einen Aufschluß über den Zweck der angesprochenen Routine geben.

Diese Art der Labeldefinition kann auch für Speicherzellen, die Daten aufnehmen sollen, benutzt werden. Beispielsweise ließe sich die Adresse der linken, oberen Ecke des Bildschirms in einem Label definieren.

```
LIOBEN   =    1024        ;LABEL FÜR DEN ZEICHENSPEICHER
FBLIOBEN =    55296       ;LABEL FÜR DEN FARBSPEICHER
         LDA  #81         ;EINEN KREIS IN DIE LINKE OBERE
         STA  LIOBEN      ;ECKE SCHREIBEN
         LDA  #14         ;DIE FARBINFORMATION IN DIE
         STA  FBLIOBEN    ;LINKE OBERE ECKE
         LDA  #42         ;EINEN STERN IN DEN
         STA  LIOBEN+1    ;ZEICHENSPEICHER SCHREIBEN
         LDA  #1          ;DIE FARBINFORMATION IN DEN
         STA  FBLIOBEN+1  ;FARBSPEICHER SCHREIBEN
         RTS              ;DAS PROGRAMM WIEDER VERLASSEN
```

Dieses Programm gibt im 40-Zeichen-Bildschirm in der linken, oberen Ecke einen gelben Kreis und einen weißen Stern aus. Zum Speichern der Zeichen und der Farbinformation wurden nicht mehr die direkten Adressen, sondern Labels verwendet. Für die linke obere Ecke des Zeichenspeichers LIOBEN als Abkürzung für "LInks OBEN", für den Farbspeicher FBLIOBEN als Abkürzung von "FarBe LInks OBEN".

Mit etwas Programmiererfahrung bekommt man ein Gespür für derartige Abkürzungen. Listings werden schneller verstanden, schließlich ist z.B. ein Name wie TABELLE leichter verständlich als der entsprechende Zahlenwert. Sie sollten sich angewöhnen , für wichtige Adressen in Ihrem Programm Labels

einzusetzen, z.B. START für den Programmanfang, SCHLEIFE für einen Schleifenanfang oder BSOUT für die entsprechende ROM-Routine. Wenn Sie die gleichen Namen in all Ihren Programmen beibehalten, werden Sie monatealte Programme sofort wieder verstehen. Wieder zurück zum Beispielprogramm. Eine weitere Neuheit ist dort eingebaut.

Anstatt für das zweite Zeichen, das rechts neben dem ersten erscheinen soll, ein weiteres Label zu definieren, werden diese Adressen errechnet. Im Bildschirmspeicher sind die Zeichen direkt aufeinanderfolgend angeordnet.

Wird zur Adresse der linken oberen Ecke, die durch das Label LIOBEN definiert ist, eins addiert, erhält man die Adresse des zweiten Zeichens auf dem Bildschirm. Entsprechend wird mit dem Farbspeicher verfahren. Diese Methode der Adressenberechnung kann die Lesbarkeit eines Programms in einigen Fällen erhöhen. Lesen Sie beispielsweise LIOBEN+1, wissen Sie sofort, daß das Zeichen rechts neben dem in der Ecke befindlichen angesprochen wird.

Noch ein Wort zu Labels: Diese dürfen nur einmal im Programm definiert werden, d.h. der Labelwert kann im Programm nicht verändert werden (ein spezieller Befehl erlaubt dies in Ausnahmesituationen).

Bei der späteren Verwendung der Labels im Operanden und in mathematischen Ausdrücken sind Labels mit vom BASIC her bekannten Variablen vergleichbar.

Aufgabe 4.2 Ein Programm soll Ihren Nachnamen auf dem 40- und 80-Zeichen-Bildschirm ausgeben. Verwenden Sie an geeigneten Stellen Labels.

Das Statusregister

JMP und JSR führen einen Sprung immer bei Erreichen des Befehls aus. Sie können nicht auf eine Programmsituation reagieren und den Sprung bedingt ablaufen lassen. Beide Befehle werden deshalb auch unter der Kategorie "unbedingte Sprünge" eingeordnet.

Der Prozessor 8502 kennt aber auch bedingte Sprünge, die auf acht verschiedene Programmsituationen reagieren können. Die Anzeige dieser Situationen ist im sogenannten Statusregister (SR) gespeichert. Dieses umfaßt 8 Bits, von denen 7 benutzt werden. Jedes Bit signalisiert einen bestimmten Zustand:

0: Der Zustand ist nicht eingetreten.
1: Der Zustand ist eingetreten.

Die Statusbits werden mit einem englischen Wort auch Statusflags genannt, da sie wie ein Signallämpchen auf etwas hinweisen. Die Bezeichnung Flags wird auch oft in Programmdokumentationen für Variablen mit ähnlichem Zweck verwendet. So zeigt ein Fehlerflag das Auftreten eines Fehlers an. Im Statusregister sind die Prozessorflags wie folgt angeordnet:

Bit	7	6	5	4	3	2	1	0
Flag	N	V	–	B	D	I	Z	C

Die Buchstaben sind die abgekürzten Namen der einzelnen Flags. In der nun folgenden Beschreibung werden die Aufgabe des Flags und die Befehle, die es beeinflussen und die einen bedingten Sprung auslösen, genannt.

N – Bit 7 – Negativ-Flag (Negativ flag)

Dieses Flag wird gesetzt, sobald das Ergebnis eines Befehls negativ, und entsprechend gelöscht, wenn das Ergebnis positiv ist. Egal, ob die Operation mit dem Akkumulator, dem X- oder Y-Register oder dem Speicher durchgeführt wird, das Flag wird immer entsprechend gesetzt.

Sie werden sich sicher fragen, wie negative Zahlen dargestellt werden können, wenn alle Register und Speicherzellen nur Werte von 0 bis 255 aufnehmen können. Eine genaue Antwort erhalten Sie in Kapitel 6.

Vorweg sei nur gesagt, daß einfach die Zahlen von 0 bis 127 als positive Zahlen definiert werden und die Zahlen von 128 bis 255 als negative. Die Befehle BPL (Branch if PLus) und BMI (Branch if MInus) führen eine Verzweigung entsprechend dem Zustand des Flags aus. Verändert werden die Flags durch folgende Befehle:

ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA, TXS, TYA

V – Bit 6 – Überlauf-Flag (oVerflow flag)

Das Überlauf-Flag wird gesetzt, wenn das Ergebnis einer mathematischen Operation des Akkumulators nicht dessen Inhalt entspricht, z.B. ein negatives Ergebnis angezeigt wird, obwohl das korrekte Ergebnis positiv wäre. Auch dieser Sachverhalt wird in Kapitel 6 genau erklärt. Die Befehle BVC (Branch if oVerflow is Clear) und BVS (Branch if oVerflow is Set) führen einen Sprung entsprechend dem Zustand des V-Flags durch.

Die folgenden Befehle beeinflussen das Flag:

ADC, BIT, CLV, PLP, RTI, SBC

Speziell sei auf den Befehl CLV (CLear oVerflow) hingewiesen, mit dem das Überlauf-Flag gezielt gelöscht werden kann.

– – Bit 5 – unbenutzt

B – Bit 4 – Break-Flag (Break-flag)

Dieses Flag weist darauf hin, daß eine Unterbrechung (Interrupt) durch die Software (BRK-Befehl) und nicht durch die angeschlossene Hardware erzeugt wurde. Alles über Interrupts erfahren Sie in Kapitel 8.

D – Bit 3 – Dezimal-Flag (Decimal mode)

Die Arithmetikbefehle ADC und SBC des Prozessors 8502 können nicht nur Binärzahlen, sondern auch Dezimalzahlen verarbeiten. Mit diesem Flag wird dem Prozessor mitgeteilt, in welchem Modus er arbeiten soll, der Binär- oder Dezimalzahlverarbeitung.

Programmbeispiele für die Dezimalbetriebsart finden Sie in Kapitel 6. Die Befehle

CLD, PLP, RTI, SED

verändern das Flag. Dieses kann gezielt mit dem Befehl CLD (CLear Decimal-mode) gelöscht und mit SED (SEt Decimal-mode) gesetzt werden.

I – Bit 2 – Unterbrechungs-Flag (Interrupt mask)

Dieses Flag sperrt eine Unterbrechung (Interrupt) durch die Hardware. Ist dieses Flag gesetzt, wird das laufende Programm nicht unterbrochen. Genaueres erfahren Sie in einem gesonderten Kapitel. Beeinflußt wird das Flag durch folgende Befehle:

BRK, CLI, PLP, RTI, SEI

Auch hier kann das Flag wieder gezielt gelöscht und gesetzt werden. Die Befehle CLI (CLear Interrupt mask) und SEI (SEt Interrupt mask) übernehmen diese Funktion.

Z – Bit 1 – Null-Flag (Zero flag)

Dieses Flag wird gesetzt, wenn das Ergebnis eines Befehls null ist. Dies kann ein Lade- oder Arithmetikbefehl sein und sich auf den Akkumulator, X- oder Y-Register oder den Speicher beziehen. Die Befehle BNE (Branch if Not Equal to zero) und BEQ (Branch if EQual to zero) verzweigen entsprechend dem Zustand des Flags. Beeinflußt wird das Flag durch die Befehle:

ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LDY, LSR, ORA, PLA, PLP, ROL, RTI, SBC, TAX, TAY, TSX, TXA, TXS, TYA

C – Bit 0 – Übertrags-Flag (Carry flag)

Das Übertrags-Flag zeigt an, ob bei einer arithmetischen Operation ein Übertrag oder Borgen erforderlich wurde, z.B. erhält man bei der Addition der Zahlen 123 und 200 das Ergebnis 323. Da der Akkumulator aber nur Zahlen bis 255 (8 Bits) aufnehmen kann, werden die untersten acht Bits des Ergebnisses (67) im Akkumulator behalten und das Übertrags-Flag gesetzt.

Dies zeigt jetzt an, daß bei der Operation ein neuntes Bit entstanden ist, das von einem höherwertigen Byte verarbeitet werden müßte. Genaueres erfahren Sie in Kapitel 6.

Dort werden auch einige Beispiele gegeben. Das Flag wird außerdem von den Schiebe- und Rotierbefehlen verwendet. Das Flag muß dort die Aufgabe eines 9. Datenbits übernehmen. Einen Sprung entsprechend dem Zustand des Flags führen die Befehle BCC (Branch if Carry Clear) und BCS (Branch if Carry Set) aus.

Verändert wird das Flag durch die Anweisungen:

ADC, ASL, CMP, CLC, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC

Gezielt beeinflußt werden kann das Statusbit mit den Befehlen CLC (CLear Carry) und SEC (SEt Carry), die Sie bereits in Kapitel 3 kennengelernt haben. In der gesamten Aufstellung über Flags dürfen die Befehle, die kein Flag beeinflussen, natürlich nicht fehlen. Diese gliedern sich in zwei Gruppen. Die ersten Befehle sind alle Sprunganweisungen, zu denen die Befehle

JMP, JSR, RTS, BPL, BMI, BVC, BVS, BNE, BEQ, BCC, BCS

gehören. Die zweite Gruppe bilden alle Befehle, die Daten in den Speicher schreiben. Die Anweisungen lauten wie folgt:

STA, STX, STY, PHA, PHP

Dies ist eine sehr wichtige Eigenschaft der Speicherbefehle. Registerinhalte können im Speicher abgelegt werden, ohne die Flags zu beeinflussen. Das Statusregister kann so über mehrere Befehle hinweg unverändert bleiben.

Alle Befehle, die ein Statusflag gezielt beeinflussen, können auf eine einfache Merkregel zurückgeführt werden. Die Befehle zum Löschen der Flags beginnen mit "CL" (CLear) gefolgt von der Abkürzung des Flags (CLC, CLD, CLI, CLV).

Die Befehle zum Setzen der Flags beginnen mit "SE" (SEt) und wieder dem Buchstaben für das Flag (SEC, SED, SEI).

Zu beachten ist, daß kein Befehl existiert, der das Überlauf-Flag gezielt setzt. Leider gibt es keine derartige Regel für die bedingten Sprünge. Sie können sich nur einprägen, daß alle mit einem "B" anfangen, was für "Branch" (deutsch: Verzweigung) steht.

Bedingte Sprünge

Nach den Vorreden zu den eigentlichen Sprungbefehlen: Bei diesen Befehlen verlangt der Prozessor im Operanden nicht die Zieladresse des Sprungs, sondern eine Angabe, wie viele Bytes vorwärts bzw. rückwärts von der aktuellen Position des Programmzeigers dieses Ziel liegt.

Da nicht die absolute Speicheradresse angegeben wird, werden diese Sprungbefehle auch relative Sprünge genannt. Wie wird der Operand des Befehls jetzt berechnet? Hierzu ein kleiner Programmausschnitt. Am Zeilenanfang steht jeweils die Speicheradresse des Befehls.

```
        .
        .
5000    SBC    #100
5002    BEQ    2
5004    LDA    #10
5006    STA    1024
        .
        .
```

Am Ende des BEQ-Befehls zeigt der Programmzeiger auf den LDA-Befehl, also auf die Adresse 5004. Wird kein Sprung ausgeführt (Ergebnis nicht null), setzt der Prozessor die Arbeit mit dem LDA-Befehl fort. Wird ein Sprung durchgeführt (Ergebnis null), addiert der 8502 die relative Adresse 2 zum Programmzeiger, also 5004 plus 2, womit dieser auf die Adresse 5006 zeigt,

den STA-Befehl. Ab diesem Befehl arbeitet der Prozessor weiter. Der Befehl LDA #10 wird also entsprechend dem Zustand des Null-Flags übersprungen. Jetzt ein Beispiel für einen Rückwärtssprung:

```
        .
        .
5000  LDX    #10
5002  DEX
5003  BNE    253
5005  RTS
        .
        .
```

Auch bei diesem Programm zeigt der Programmzeiger am Ende des BNE-Befehls auf den nachfolgenden Befehl. Um jetzt den DEX-Befehl zu erreichen, muß der Programmzeiger um 3 vermindert werden. Der Operand von BNE müßte also -3 lauten.

Da der Prozessor eine derartige Zahl nicht versteht, wird statt dessen der Wert 256 – 3, also 253, eingesetzt. Um diesen Wert von den Vorwärtssprüngen unterscheiden zu können, wurde festgelegt, daß die Zahlen 0 bis 127 einen Vorwärtssprung bewirken und die restlichen Zahlen, größer 127, einen Rückwärtssprung.

An der Größe der Zahlenwerte erkennen Sie auch, daß maximal 127 Bytes vorwärts bzw. rückwärts übersprungen werden können. Aus diesen Zahlenwerten ist auch zu erkennen, daß der Operand 1 Byte lang ist. Sie merken schon, daß diese Sprungberechnung für die Assemblerprogrammierung sehr unhandlich ist.

Wird z.B. ein neuer Befehl eingefügt, der zusätzlich übersprungen werden soll, so ändert sich auch der Operand des Sprungbefehls, da jetzt mehr Befehle übersprungen werden müssen. Dieser müßte neu berechnet werden. Zudem gestaltet sich die Bestimmung der Adressendistanz bei langen Sprüngen sehr schwierig.

Hier unterstützt wieder der Assembler den Programmierer. Statt der Sprungweite wird im Assemblerprogramm die Zieladresse des Sprungs eingegeben. Der Assembler berechnet hieraus die Adressendistanz für den Verzweigungsbefehl.

Wie bei JMP- und JSR-Befehlen kann natürlich ein Label als Zieladresse Verwendung finden. Das erste Beispiel würde, mit Labels aufgebaut, wie folgt aussehen:

```
          .
          .
          SBC
#100
          BEQ       STORE
          LDA       #10
STORE     STA       1024
          .
          .
```

Beim BEQ-Befehl holt der Assembler den Wert des Labels STORE, also die Adresse des STA-Befehls, und berechnet die Sprungweite. Der Programmierer erkennt natürlich bei dieser Art der Sprungvorgabe nicht, wann die maximale Sprungdistanz von 127 Bytes vorwärts bzw. rückwärts erreicht ist.

Erst der Assembler stellt dies bei der Berechnung fest. Er gibt deshalb auch eine Fehlermeldung aus, sobald ein zu großer Sprung aufgetreten ist. Die Meldung lautet entsprechend:

```
?ZU GROSSER RELATIVER SPRUNG
```

Da jeder Befehl im Durchschnitt 2 Bytes belegt, können Sie sich als Regel merken, daß etwa 60 Programmzeilen maximal übersprungen werden können. Um einen trotzdem aufgetretenen Fehler wieder zu beseitigen, umschreiben Sie den Branch-Befehl durch einen JMP-Befehl. Dies könnte wie folgt aussehen:

```
          .                                   .
          .                                   .
          SBC    #100                         SBC      #100
          BEQ    ENDE                         BNE   WEITER
          .                                   JMP      ENDE
          .                           WEITER  .
          .                                   .
ENDE      RTS                         ENDE             RTS
```

Auf der linken Seite wird zur Marke ENDE verzweigt, wenn das Ergebnis null ist. Auf der rechten Seite wird durch den Branch-Befehl zum weiterführenden Ast verzweigt und durch einen JMP-Befehl zur Marke ENDE gesprungen.

Bei dieser Konstruktion darf die Zieladresse ENDE beliebig weit entfernt sein. Diese Programmierung benötigt aber auch eine Anweisung bzw. 3 Bytes (die Länge des JMP-Befehls im Speicher) mehr. Sie sollten einen Branch-Befehl immer einem JMP-Befehl vorziehen.

Schleifen

Die bedingten Sprünge sollen jetzt in Anwendungen genau beschrieben werden.

BNE – Branch if Not Equal to zero (Z=0)
– Verzweige, wenn ungleich null

BEQ – Branch if EQual to zero (Z=1)
– Verzweige, wenn gleich null

Beide Befehle führen einen Sprung in Abhängigkeit vom Null-Flag aus. Bei Vergleichsoperationen und Schleifen liegt das Hautanwendungsgebiet. Eine Schleife soll auch im ersten Beispiel aufgebaut werden. In Abb. 3 sehen Sie das dazugehörige Flußdiagramm.

```
BSOUT     =          $FFD2
;
          LDX   #10     ;X-REGISTER IST SCHLEIFENZÄHLER
          LDA   #"*"    ;CODE EINES STERNS IN DEN AKKUMULATOR
SCHLEIFE  JSR   BSOUT   ;DEN AKKUMULATOR-INHALT AUSGEBEN
          DEX           ;SCHLEIFENZÄHLER UM EINS VERMINDERN
          BNE  SCHLEIFE;SCHLEIFENZÄHLER=0 ?,NEIN =>
          RTS           ;PROGRAMM WIEDER VERLASSEN
```

Geben Sie das Programm ein, und beobachten Sie das Ergebnis. Auf dem Bildschirm erscheinen 10 Sterne. Übrigens arbeitet das Beispiel, wie Sie sicher schon erkannt haben, im 40- und 80-Zeichen-Modus des C128.

Ab sofort gilt dies für alle weiteren Beispiele. Die Arbeit des Assemblers soll wieder im Assemblerlisting betrachtet werden:

```
                    0001              .OBJ   M
                    0002              .BANK  15
                    0003              *=     $1300
                    0004    BSOUT     =      $FFD2
                    0005    ;
F1300  A2 0A        0006    LDX       #10              ;X-REGISTER IS...
F1302  A9 2A        0007    LDA       #"*"             ;CODE EINES ST...
F1304  20 D2 FF     0008    SCHLEIFE  JSR    BSOUT     ;DEN AKKU.-INHA...
F1307  CA           0009              DEX              ;SCHLEIFENZÄH...
F1308  D0 FA        0010              BNE    SCHLEIFE  ;SCHLEIFENZÄH...
F130A  60           0011              RTS              ;PROGRAMM WIED...
F130B-ENDE DER ASSEMBLIERUNG!
```

Auch hier erkennt man die Leistung des Assemblers wieder. Er erkennt den Branch-Befehl und berechnet die Sprungdistanz. Beim Unterprogrammaufruf wird die korrekte Zieladresse in den Objektcode eingebaut.

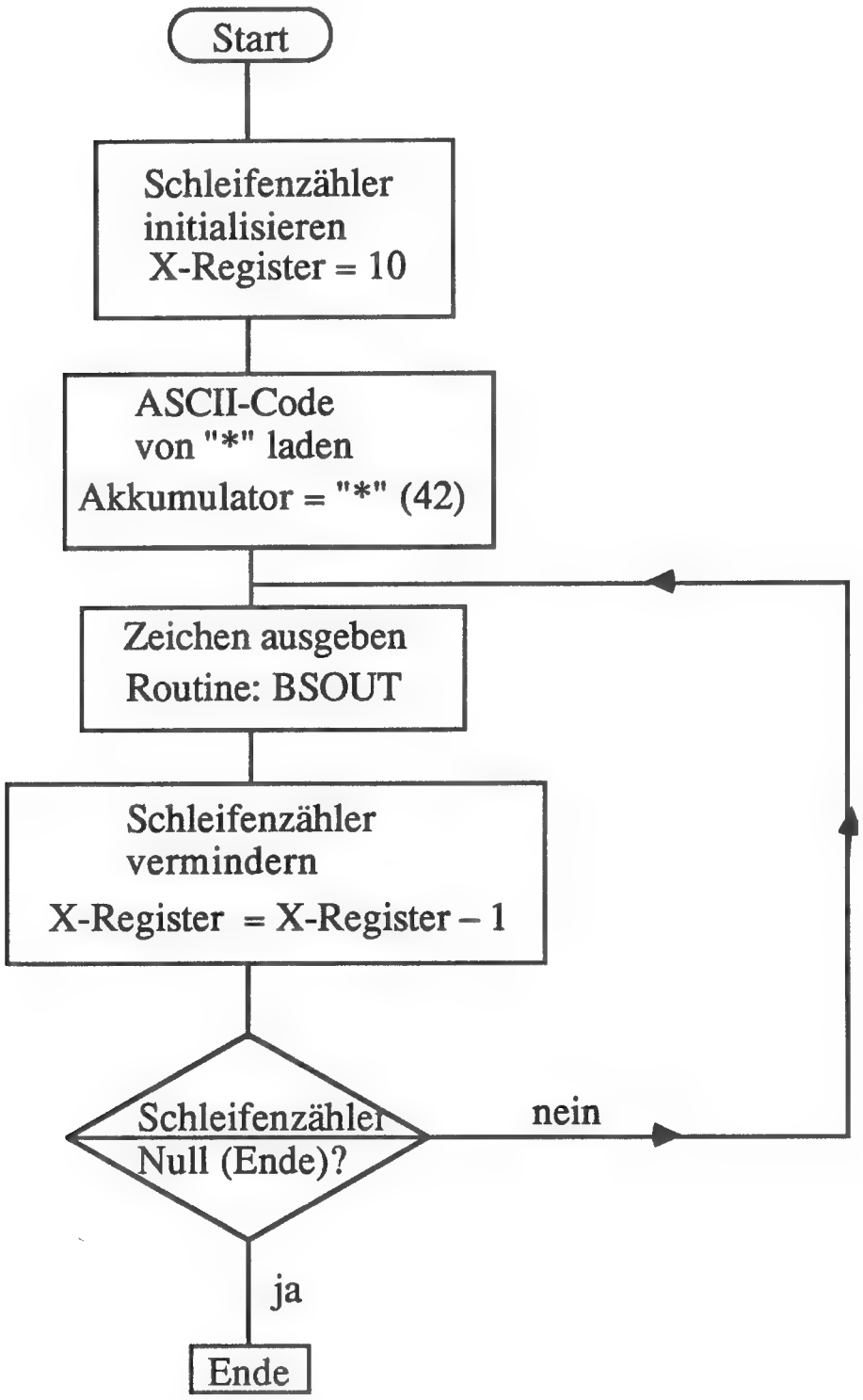


Abb. 4.1: X-Register als Schleifenzähler

Das Beispiel verwendet das X-Register als Schleifenzähler. Dieser wird in der zweiten Zeile initialisiert, indem die Zahl 10 geladen wird. Der Schleifenzähler wird durch Herunterzählen verändert.

Das Endkriterium der Schleife ist erreicht, wenn das X-Register nach dem Dekrementieren den Wert Null enthält. Ist diese Bedingung nicht erfüllt, verzweigt der BNE-Befehl wieder zum Schleifenanfang. Da als Startwert ins X-Register 10 geladen wurde, muß das X-Register zehnmal dekrementiert werden, bis das Endkriterium erreicht ist. Der Schleifenrumpf wird also auch

zehnmal durchlaufen. Für die Ausgabe wird bei der Initialisierung der Akkumulator mit dem ASCII-Code eines Sterns geladen. In der Schleife wird dieser mit der Routine BSOUT auf dem Bildschirm dargestellt. Beim nächsten Durchlauf braucht der Akkumulator nicht mehr geladen zu werden, da die Ausgaberoutine den Inhalt des Akkumulators nicht verändert.

Der gesamte Programmablauf soll zur Verdeutlichung in einer Tabelle dargestellt werden. In der Spalte Z wird die Entwicklung des Null-Flags aufgezeigt. Nach der BSOUT-Routine ist der Zustand undefiniert, was durch ein Fragezeichen dargestellt wird.

```
Programm:          PC     AC     XR    Z     Ausgabe

LDX      #10       4864   ???    10    0     -
LDA      #"*"      4866   42     10    0     -
JSR      BSOUT     4868   42     10    ?     *
DEX                4871   42     9     0     -
BNE      SCH...    4872   42     9     0     -

JSR      BSOUT     4868   42     9     ?     *
DEX                4871   42     8     0     -
BNE      SCH...    4872   42     8     0     -
JSR      BSOUT     4868   42     8     ?     *
 .
JSR      BSOUT     4868   42     1     ?     *
DEX                4871   42     0     1     -
BNE      SCH...    4872   42     0     1     -
RTS                4873   42     0     1     -
```

Die Tabelle zeigt deutlich die Zusammenhänge zwischen DEX-Befehl, dem Null-Flag und der BNE-Verzweigung. Solange das X-Register einen Wert ungleich null enthält, wird zum Schleifenanfang verzweigt. Die Tabelle zeigt auch, daß das Null-Flag erst durch eine Operation entsprechend dem Inhalt des X-Registers gesetzt wird.

Direkt im Anschluß an die BSOUT-Routine kann das Schleifenende nicht abgefragt werden, da das Null-Flag durch Anweisungen in der BSOUT-Routine einen undefinierten Zustand besitzt. Durch Mißachten dieser Vorgänge, was auch für andere Flags gilt, schleichen sich häufig Fehler ins Programm ein.

Es ist deshalb wichtig zu wissen, in welchem Zustand sich die Flags an jeder Programmposition befinden. Man erspart sich viel Ärger bei einer Fehlersuche. Eine weitere Eigenschaft des Null-Flags ist aus der Tabelle ersichtlich. Das Flag wird gesetzt, wenn das Ergebnis null ist, und gelöscht, wenn es ungleich null ist, also immer genau entgegengesetzt. An diese paradoxe Eigenschaft des Flags muß man sich erst gewöhnen.

Aufgabe 4.3 Schreiben Sie ein Programm, das 5 Kommata und in der nächsten Zeile 8 Dezimalpunkte ausgibt. Verwenden Sie im Programm sowohl das X- als auch das Y-Register zum Aufbau von Schleifen.

Es sollen die Sterne aus dem Beispielprogramm jetzt langsam hintereinander auf dem Bildschirm erscheinen. In das bisherige Programm muß eine Verzögerung eingebaut werden. Eine Schleife, die nichts tut, also nur Rechenzeit vom Prozessor benötigt, erfüllt genau diesen Zweck. Das bisherige Programm verwendet nur den Akkumulator und das X-Register, eine weitere Schleife kann also mit dem Y-Register aufgebaut werden. Das erweiterte Programm sieht wie folgt aus:

```
BSOUT  =      $FFD2
;
              LDX  #10          ;X-REGISTER IST SCHLEIFENZAEHLER
              LDA  #"*"         ;CODE EINES STERNS IN DEN AKKU.
SCHLEIFE      JSR  BSOUT        ;DEN AKKU.-INHALT AUSGEBEN
;
              LDY  #255         ;WARTESCHLEIFE AUF STARTWERT
WARTEN        DEY               ;DEN ZAEHLER VERMINDERN
              BNE  WARTEN       ;SCHLEIFENZAEHLER=0 ?,NEIN =>
;
              DEX               ;SCHLEIFENZAEHLER UM EINS VERMINDERN
              BNE  SCHLEIFE     ;SCHLEIFENZAEHLER=0 ?,NEIN =>
              RTS               ;PROGRAMM WIEDER VERLASSEN
```

Nach der Eingabe werden Sie sicher enttäuscht feststellen, daß die Sterne immer noch mit rasanter Geschwindigkeit ausgegeben werden. Die Verzögerung durch die Schleife ist noch nicht groß genug. Diese müßte nochmals in eine Schleife gepackt werden, um die Verzögerungszeit zu verlängern.

Das Programm belegt aber bereits alle freien Register. Diese können also nicht als Schleifenzähler herangezogen werden. Der Einsatz des Speichers wird notwendig. Eine Speicherzelle muß als Schleifenzähler eingesetzt werden. Im Befehlssatz des Prozessors 8502 werden durch Inkrementier- und Dekrementierbefehle für Speicherzellen bereits die Voraussetzungen gegeben. Die Befehle lauten:

INC – INCrement memory
– inkrementiere den Speicher

DEC – DECrement memory
– dekrementiere den Speicher

Die Adresse der Speicherzelle wird im Operanden angegeben. Die Statusflags werden wie bei den anderen Inkrementier- und Dekrementierbefehlen beeinflußt. Auch das Verhalten bei Erreichen der Zahlgrenzen 0 und 255 ist

identisch. Eine Schleife mit einer Speicherzelle könnte jetzt wie folgt aussehen:

```
ZAEHLER      =    5000
             .
             .
             LDA  #20
             STA  ZAEHLER
SCHLEIFE     DEC  ZAELER
             BNE  SCHLEIFE
             .
             .
```

Als Schleifenzähler wird die Speicherzelle 5000 definiert. Für das Programm tritt stellvertretend das Label ZAEHLER in Erscheinung. Der Vorteil von Labels wird hier wieder deutlich. Soll eine andere Speicherzelle als Zähler zum Einsatz kommen, braucht nur die Wertzuweisung an das Label ZÄHLER geändert zu werden und nicht jede Programmzeile, die auf den Zähler zugreift. Das vollständige Programm mit erweiterter Verzögerungschleife sieht jetzt wie folgt aus:

```
ZAEHLER        =    5000
BSOUT          =    $FFD2
;
LDX            #10          ;X-REGISTER IST SCHLEIFENZAEHLER
LDA            #"*"         ;CODE EINES STERNS IN DEN AKKU.
SCHLEIFE       JSR  BSOUT    ;DEN AKKU.-INHALT AUSGEBEN
;
               LDY  #100     ;AEUSSEREN SCHLEIFENZAEHLER MIT
               STY  ZAEHLER  ;STARTWERT LADEN
WARTEN1        LDY  #255     ;INNERE SCHLEIFE MIT STARTWERT LADEN
WARTEN2  DEY                 ;DEN INNEREN ZAEHLER VERMINDERN
               BNE  WARTEN2  ;INNERER ZAEHLER=0 ?,NEIN =>
               DEC  ZAEHLER  ;DEN AEUSSEREN ZAEHLER VERMINDERN
               BNE  WARTEN1  ;AUESSERER ZAEHLER=0 ?,NEIN =>
;
               DEX           ;SCHLEIFENZAEHLER UM EINS VERMINDERN
               BNE  SCHLEIFE ;SCHLEIFENZAEHLER=0 ?,NEIN =>
               RTS           ;PROGRAMM WIEDER VERLASSEN
```

Die Sterne erscheinen langsam nacheinander auf dem Bildschirm. Die äußere Warteschleife hat mit 100 noch nicht ihren Maximalwert. Die Ausgabe kann nochmals um das 2,5-fache verlangsamt werden. Sicher wird es Sie interessieren, wie die Verzögerungszeit der Schleifen berechnet wird. Hierzu ein kleiner Ausflug in die innere Arbeitsweise des Prozessors.

Die gesamten Abläufe im Computer werden durch den sog. Systemtakt koordiniert. Er bestimmt, wann auf den Speicher zugegriffen wird, wann der Prozessor seinen nächsten Arbeitsschritt macht usw. Im SLOW-Modus des

C128 beträgt die Taktrate ein MHz, also 1 Million Steuerimpulse in der Sekunde. Im FAST-Modus verdoppelt sich die Geschwindigkeit auf zwei MHz. In Befehlsübersichten, wie sie auch in Anhang C zu finden sind, wird die von jedem Befehl benötigte Zahl von Takten angegeben. INX verbraucht beispielsweise zwei Taktzyklen und BNE drei Zyklen. Hier die Taktzahl aller Befehle der Verzögerungsschleife:

Der Rumpf der inneren Schleife benötigt 2+3=5 Takte und mit ihren 255 Durchläufen also 5*255=1275. Der Rumpf der äußeren Schleife benötigt entsprechend 2+1275+6+3=1286 und bei 100 Durchläufen 1286*100=128600 Takte.

Hinzu kommen noch sechs Takte zur Initialisierung der Schleife, womit man 128 606 Takte als Endergebnis erhält. Zur Berechnung der benötigten Zeit in Sekunden, muß die Taktzahl durch die Taktfrequenz dividiert werden.

SLOW-Modus: 128606/1000000 = 0.129
FAST-Modus: 128606/2000000 = 0.064

Dies ist nur eine ungefähre Zeit. Viele Faktoren spielen eine Rolle. So benötigten die Branch-Befehle einen Taktzyklus weniger, wenn keine Verzweigung stattfindet, oder jede 60stel-Sekunde wird automatisch die Tastatur abgefragt. Die errechnete Zeit gibt jedoch einen guten Anhaltspunkt. Das Beispielprogramm soll ein letztes Mal erweitert werden. Es sollen nicht mehrere Sterne erscheinen, sondern ein Stern soll sich über den Bildschirm bewegen. Realisiert wird dies, indem der Stern erst gezeichnet, dann der Cursor zurückgesetzt und der Stern wieder gelöscht wird. Die ganzen Operationen werden durch den ASCII-Code gesteuert. Das Programm sieht wie folgt aus:

```
ZAEHLER =  5000
BSOUT   =  $FFD2
;
           LDX  #10      ;X-REGISTER IST SCHLEIFENZAEHLER
SCHLEIFE   LDA  #157     ;CODE VON CURSOR-LINKS IN DEN AKKU
           JSR  BSOUT    ;DEN CODE AUSGEBEN
           LDA  #" "     ;CODE EINER LEERSTELLE IN DEN AKKU.
           JSR  BSOUT    ;UND DEN LETZTEN STERN DAMIT LOESCHEN
           LDA  #"*"     ;CODE EINES STERNS IN DEN AKKU
           JSR  BSOUT    ;DEN AKKU.-INHALT AUSGEBEN
```

```
;
            LDY  #100      ;AEUSSEREN SCHLEIFENZAEHLER MIT
            STY  ZAEHLER   ;STARTWERT LADEN
WARTEN1     LDY  #255      ;INNERE SCHLEIFE MIT STARTWERT LADEN
WARTEN2     DEY            ;DEN INNEREN ZAEHLER VERMINDERN
            BNE  WARTEN2   ;INNERER ZAEHLER=0 ?,NEIN =>
            DEC  ZAEHLER   ;DEN AEUSSEREN ZAEHLER VERMINDERN
            BNE  WARTEN1   ;AUESSERER ZAEHLER=0 ?,NEIN =>
;
            DEX            ;SCHLEIFENZAEHLER UM EINS VERMINDERN
            BNE SCHLEIFE   ;SCHLEIFENZAEHLER=0 ?,NEIN =>
            RTS            ;PROGRAMM WIEDER VERLASSEN
```

Als Ergebnis wandert ein Stern um zehn Zeichen nach rechts. Bei der Eingabe des Programms ist Ihnen sicherlich aufgefallen, daß beim ersten Aufruf der Hauptschleife ein Stern gelöscht wird, der gar nicht existiert. Würde das Löschen nach der Verzögerungsschleife stattfinden, wäre dieser Umstand beseitigt. Jetzt würde jedoch der letzte gezeichnete Stern am Ende des Programms vom Bildschirm verschwinden, was auch nicht wünschenswert ist. Probieren Sie diese Änderung doch einmal aus.

Aufgabe 4.4 Ein Programm soll einen Kreis über 15 Zeilen hinweg langsam von oben nach unten bewegen. Variieren Sie auch die Bewegungsgeschwindigkeit. Versuchen Sie, das Programm vollkommen selbständig zu erstellen.

Verschiedene Schleifenarten

Die im letzten Abschnitt behandelten Schleifen gehören zur Kategorie FOR...NEXT-Schleifen, die Sie sicher auch von BASIC her kennen. Der Schleifenzähler wurde durch Dekrementieren an die Endbedingung herangeführt. Der Zähler der Schleife soll jetzt durch Inkrementieren verändert werden.

Bei der Behandlung der INX/Y- und DEX/Y-Befehle haben Sie das Verhalten der Register bei Erreichen der Zahlgrenzen kennengelernt. Dieser Effekt kann für den Aufbau einer aufwärtszählenden Schleife ausgenutzt werden. Der Schleifenzähler zählt bis zum Wert 255, und beim nächsten Inkrement enthält er den Wert Null.

Dieser Punkt wird mit dem BEQ- bzw. BNE-Befehl abgefragt. Das Programm zur Ausgabe von zehn Sternen wird auf diese Schleifenart umgeschrieben. Zur Abwechslung wird als Schleifenzähler das Y-Register eingesetzt.

```
BSOUT   =       $FFD2
;
            LDY  #246          ;Y-REGISTER IST SCHLEIFENZAEHLER
            LDA  #"*"          ;CODE EINES STERNS IN DEN AKKU
SCHLEIFE    JSR  BSOUT         ;DEN AKKU.-INHALT AUSGEBEN
            INY                ;SCHLEIFENZAEHLER UM EINS ERHOEHEN
            BNE  SCHLEIFE      ;SCHLEIFENZAEHLER=0 ?,NEIN =>
            RTS                ;PROGRAMM WIEDER VERLASSEN
```

Im Ergebnis ist kein Unterschied zur ursprünglichen Programmversion zu sehen.

Aufgabe 4.5 Durch eine kleine Abwandlung des letzten Programmbeispiels können die ASCII-Codes von 246 bis 255 ausgegeben werden. Suchen Sie diese Möglichkeit, und ändern Sie das Programm.

Sehr befriedigend sind diese Schleifenarten noch nicht. Es sollte z. B. auch möglich sein, einen Schleifenzähler von 65 bis 90 laufen zu lassen. Mit einem etwas umständlichen Weg ließe sich dies wie folgt lösen:

```
            LDX  #65
SCHLEIFE    .
            .
            INX
            TXA
            SEC
            SBC  #91
            BNE  SCHLEIFE
            RTS
```

In dieser Konstruktion wird das X-Register als Schleifenzähler verwendet und durch Inkrementieren an die Endbedingung herangeführt. Um diese abzufragen, wird vom Inhalt des X-Registers 91 subtrahiert (über den Akkumulator).

Erst wenn diese Rechnung null ergibt, also das X-Register den Wert 91 enthält, ist die Endbedingung erfüllt. Wieso muß eigentlich das Register den Wert 91 enthalten und nicht 90, bis zu dem die Schleife laufen soll?

Im Programmtext sehen Sie, daß der INX-Befehl erst nach dem Schleifenrumpf eingebaut ist. Nachdem dieser mit dem Wert 90 im X-Register durchlaufen wurde, erfolgt also nochmals ein Inkrement. Als Endbedingung muß das X-Register deshalb 91 enthalten.

Der im Beispiel beschriebene Weg ist zwar gangbar, aber sehr umständlich, da viele Befehle benötigt werden und der Inhalt des Akkumulators verloren geht. Drei spezielle Befehle des Prozessors 8502 erlauben es, den Inhalt der Register mit einem Wert zu vergleichen.

CMP – CoMPare memory with accumulator
– vergleiche den Operanden mit dem Akkumulator

CPX – ComPare X-register with memory
– vergleiche den Operanden mit dem X-Register

CPY – ComParY-register with memory
– vergleiche den Operanden mit dem Y-Register

Wie bei den Ladebefehlen darf im Operanden ein direkter Zahlenwert oder die Adresse einer Speicherzelle stehen. Intern führt der Prozessor bei diesen Befehlen eine Subtraktion durch. Im Unterschied zum SBC-Befehl wird das Endergebnis aber nicht im Register abgelegt, sondern es werden nur die Statusflags ent- sprechend dem Ergebnis gesetzt (nicht das V-Flag).

Außerdem muß der Prozessor bei diesem Befehl nicht mit SEC auf die Subtraktion vorbereitet werden. Dies übernimmt er selbständig. Die obige Schleife sähe mit einem dieser Befehlen wie folgt aus:

```
            LDX  #65
SCHLEIFE    .
            .
            INX
            CPX  #91
            BNE  SCHLEIFE
            RTS
```

Eingesetzt in einem kleinen Programm kann mit dieser Schleife das Alphabet auf dem Bildschirm dargestellt werden.

```
BSOUT       =    $FFD2
;
            LDX  #65      ;STARTWERT DER SCHLEIFE GLEICH 65
SCHLEIFE    TXA           ;WERT DES ZAEHLERS IN DEN AKKU.
            JSR  BSOUT    ;DESSEN INHALT AUSGEBEN
            INX           ;DEN SCHLEIFENZAEHLER ERHOEHEN
            CPX  #91      ;UND MIT 91 VERGLEICHEN
            BNE  SCHLEIFE;IST DIESER WERT ERREICHT ?,NEIN=>
            RTS           ;DAS PROGRAMM VERLASSEN
```

Bei dieser Verwendung des BNE-Befehls wird dessen Abkürzung (Branch if Not Equal) verständlicher. Der Sprung wird nur durchgeführt, wenn ein vorheriger Vergleichsbefehl einen Unterschied zwischen beiden Zahlen festgestellt hat und damit das Z-Flag löscht.

Aufgabe 4.6 Schreiben Sie das Programm so um, daß das Alphabet rückwärts auf dem Bildschirm erscheint.

Aufgabe 4.7 Ein Programm soll den ASCII-Code ausgeben. Die Steuercodes von 0-31 und 128-159 werden unterdrückt.

Bereits in zwei Aufgaben sollten Sie Ihren Namen auf dem Bildschirm darstellen. Es war immer eine aufwendige Arbeit, für jeden einzelnen Buchstaben die Ausgabebefehle einzutippen. Eine Schleife wäre für diesen Zweck ideal.

Der Schleifenrumpf könnte ein einziges Mal die Ausgaberoutine enthalten, und über den ZÄHLER werden die einzelnen Zeichen zur Ausgabe befördert. Die Programmlänge würde dann auch nicht durch die Länge des Namens beeinflußt. Schleifen können Sie jetzt sehr gut aufbauen, auch die Ausgabe bereitet Ihnen keine Schwierigkeit, aber wie werden die verschiedenen Zeichen aus dem Namen gelesen und dazu noch in Abhängigkeit vom Schleifenzähler? Für diesen Zweck kennt der Prozessor eine spezielle Form des Operanden. Hier einige Beispiele:

```
LDA  5000,X
STA  5000,X
LDX  TABELLE,Y
INC  TABELLE,X
SBC  ENDE,X
```

Jetzt nimmt der Prozessor zum Adressieren der Speicherzellen den Wert im Operanden und addiert zu diesem den Inhalt des angegebenen Registers. Enthält das X-Register beispielsweise den Wert 67 und wird der Befehl STA 5000,X bearbeitet, speichert der Prozessor den Inhalt des Akkumulators in die Speicherzelle 5000 + 67 = 5067.

In einer Anwendung kann z.B. die Operandenadresse den Anfang einer Tabelle und das X-Register einen Zeiger in die Tabelle darstellen. Daraus leitet sich auch der Name Indexregister für das X- und Y-Register ab.

Da beide Register Werte von 0 bis 255 aufnehmen können, kann auf diese Weise eine Tabelle mit 256 Elementen – 0 muß mitgezählt werden – adressiert werden. Der eigene Name kann also in einer Tabelle abgelegt und mit der neuen Operandenart ausgelesen werden.

Dieses Problem wäre gelöst. Wie wird der Name aber in eine für das Assemblerprogramm erreichbare Tabelle gebracht?

Mit dem BASIC-Befehl POKE ließen sich z.B. alle Zeichen des Namens in den Speicher schreiben. Dies ist aber sehr umständlich. Die Assembleranweisung .BYTE gestattet es, Zahlenwerte mit der Länge eines Bytes in den Objektcode einzubauen.

Die einzelnen Werte werden nach dem .BYTE-Befehl eingegeben und durch Kommata getrennt. Das Ergebnis ist deutlich im Assemblerlisting (erzeugt mit .LIST) zu erkennen.

```
  .
  .
F1373 8D 88 13  0023     STA ZAEHLER
F1376 60        0024     RTS
F1377 01 02 03  0025     .BYTE 1,2,3,4
F137A 04
F137B 64 41     0026     .BYTE 100,"A"
F137C 19 AB 03  0027     .BYTE 5*5,$AB,%101
  .
  .
```

Zur Darstellung des Namens werden die ASCII-Codes der einzelnen Buchstaben mit dem .BYTE-Befehl zu einer Tabelle zusammengefaßt. Wie bei Assemblerbefehlen kann auch der .BYTE-Befehl mit einem Label versehen werden. Die Tabelle ist so für das Assemblerprogramm erreichbar. Hier das vollständige Listing. Bei der Tabelle müssen Sie natürlich Ihren eigenen Namen eintragen.

Sollte dieser nicht in einer Zeile Platz haben, verwenden Sie einfach mehrere .BYTE-Anweisungen. Die Endbedingung der Schleife muß natürlich entsprechend der Namenslänge korrigiert werden.

```
BSOUT      =     $FFD2
;
           LDX   #0              ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE   LDA   NAME,X          ;BUCHSTABE AUS DER TABELLE LESEN
           JSR   BSOUT           ;DIESE ZEICHEN AUSGEBEN
           INX                   ;SCHLEIFENZAEHLER ERHOEHEN
           CPX   #5              ;DAS LETZTE ZEICHEN ?
           BNE   SCHLEIFE        ;NEIN =>
           RTS                   ;DAS PROGRAMM WIEDER VERLASSEN
NAME       .BYTE 83,89,66,69,88 ;NAMENSTABELLE
```

Als Schleifenzähler wird wieder das X-Register eingesetzt. Dieses beginnt die Schleife mit null. Im Schleifenrumpf wird als erstes das Zeichen aus der Tabelle gelesen. Das Label NAME steht für den Anfang der Namenstabelle. Gleichzeitig fungiert das X-Register als Index in die Tabelle, womit die verschiedenen Tabelleneinträge gelesen werden können.

Das Zeichen wird natürlich wieder mit der Routine BSOUT ausgegeben. Das X-Register, der Schleifenzähler, wird mit dem INX-Befehl aufwärtsgezählt und mit CPX auf den Endwert geprüft. Um das erste Zeichen aus der Tabelle zu erreichen, beträgt der Index 0, für das zweite Zeichen 1, für das dritte 2, usw. Also: Für das n-te Zeichen muß der Index (das X-Register) n-1 enthalten.

Zeichen	1	2	3	4	5
Name	S	Y	B	E	X
Index	0	1	2	3	4

Bei einem fünf Zeichen langen Namen muß der Schleifenrumpf als letztes mit Index = 4 durchlaufen werden. Aber wie bei den zuvor beschriebenen Schleifen muß auch hier die Endbedingung um eins größer, also hier 5, sein.

```
Programm:     PC      AC   XR   Z  Tab.-Adresse  Zeichen
LDX #0        4864    ???  0    1  -              -
LDA NAME,X    4866    83   0    0  4878+0=4878    83 / S
JSR BSOUT     4869    83   0    ?  -              -
INX           4872    83   1    0  -              -
CPX #5        4873    83   1    0  -              -
BNE SCH...    4875    83   1    0  -              -
LDA NAME,X    4866    89   1    0  4878+1=4879    89 / Y
JSR BSOUT     4869    89   1    ?  -              -
INX           4872    89   2    0  -              -
.
.
LDA NAME,X    4866    88   4    0  4878+4=4882    88 / X
JSR BSOUT     4869    88   4    ?  -              -
INX           4872    88   4    0  -              -
 CPX #5       4873    88   5    1  -              -
BNE SCH...    4875    88   5    1  -              -
RTS             4877    88   5    1  -              -

Tabelle:    4878 4879 4880 4881 4882
            S    Y    B    E    X
```

Durch eine kleine Abwandlung soll der Index der Zeichennummer und die Endbedingung der Namenslänge entsprechen. Beim ersten Aufruf des Schleifenrumpfs muß der Index auf eins stehen, damit dieser identisch mit der Zeichennummer ist.

Zu diesem Zweck startet das X-Register mit dem Wert Null, und der INX-Befehl wird an den Anfang des Schleifenrumpfs gelegt. Damit enthält der Zähler beim Lesen des ersten Zeichens den Wert Eins. Um die Zeichen aus der Tabelle auszulessen, kann der Schleifenzähler jetzt nicht mehr direkt verwendet werden. Hier greift man zu einem kleinen Trick.

Das Programm arbeitet mit einem um eins verminderten Tabellenanfang statt der tatsächlichen Adresse. Bei einem Indexwert von eins wird dann auch das erste Zeichen gelesen.

```
BSOUT       =     $FFD2
;
            LDX   #0                ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE    INX                     ;SCHLEIFENZAEHLER ERHOEHEN
```

```
            LDA     NAME-1,X        ;BUCHSTABE AUS DER TABELLE LESEN
            JSR     BSOUT           ;DIESE ZEICHEN AUSGEBEN
            CPX     #5              ;DAS LETZTE ZEICHEN?
            BNE     SCHLEIFE        ;NEIN =>
            RTS                     ;DAS PROGRAMM WIEDER VERLASSEN
NAME        .BYTE 83,89,66,69,88    ;NAMENSTABELLE
```

Auch hier wieder eine Tabelle zur Verdeutlichung der Vorgänge im Programm:

```
Programm:     PC      AC   XR   Z  Tab.-Adresse  Zeichen
LDX  #0       4864    ???  0    1  -             -
INX           4872    83   1    0  -             -
LDA NAME,X    4866    83   1    0  4877+1=4878   83 / S
JSR BSOUT     4869    83   1    ?  -             -
CPX #5        4873    83   1    0  -             -
BNE SCH...    4875    83   1    0  -             -
INX           4872    89   2    0  -             -
LDA NAME,X    4866    89   2    0  4877+2=4879   89 / Y
JSR BSOUT     4869    89   2    ?  -             -
.
.
INX           4872    89   5    0  -             -
LDA NAME,X    4866    88   5    0  4877+5=4882   88 / X
JSR BSOUT     4869    88   5    ?  -             -
CPX #5        4873    88   5    1  -             -
BNE SCH...    4875    88   5    1  -             -
RTS           4877    88   5    1  -             -

Tabelle:      4878 4879 4880 4881 4882
              S    Y    B    E    X
```

Der versetzte Tabellenanfang nutzt wieder den Vorteil der Labels aus. Die korrekte Adresse wird durch einen mathematischen Ausdruck errechnet und vom Assembler in den Objektcode eingebaut.

Welcher der beiden Schleifenformen der Vorzug zu geben ist, muß von Fall zu Fall entschieden werden. Ein bei null beginnender Index bietet jedoch durch die besondere Lage von null im Zahlenbereich häufig Vorteile (vergleiche Dekrement von null).

Die Eingabe des Namens soll jetzt verbessert werden. Der Assembler stellt eine Anweisung zur Verfügung, mit der Texte in den Objektcode übernommen werden können.

Der Befehl lautet .TEXT. Außer Strings können, wie beim .BYTE-Befehl, Zahlenwerte beigemischt werden. Einige Beispiele für den Befehl in einem Assemblerlisting:

```
F1370 4E 41 4D  0030                .TEXT "NAME"
F1373 45
F1374 53 59 42  0031                .TEXT "SYBEX",13
F1377 45 58 0D
F1378 41 55 46  0032                .TEXT "AUF",0,"ZU",0
F137B 00 5A 55
F137E 00
```

Ins Programm übernommen sieht dies wie folgt aus:

```
BSOUT       =       $FFD2
;
            LDX     #0              ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE    LDA     NAME,X          ;BUCHSTABE AUS DER TABELLE LESEN
            JSR     BSOUT           ;DIESE ZEICHEN AUSGEBEN
            INX                     ;SCHLEIFENZAEHLER ERHOEHEN
            CPX     #5              ;DAS LETZTE ZEICHEN?
            BNE     SCHLEIFE        ;NEIN =>
            RTS                     ;DAS PROGRAMM WIEDER VERLASSEN
NAME        .TEXT "SYBEX"           ;NAMENSTABELLE
```

Das Programm kann noch weiter verbessert werden. Bei der Änderung des Namens muß meist, durch die Längenänderung verursacht, die Schleifenendbedingung nachgeführt werden. Um das Problem zu lösen, wird die Endbedingung nicht in die Schleife, sondern in den Namen selbst übernommen.

Ein bestimmtes Zeichen im Namen signalisiert das Ende. Im ASCII-Code ist der Code Null für diesen Zweck reserviert (EOM = End Of Message). Nach dem Laden eines Zeichens aus der Tabelle wird mit dem BEQ-Befehl entschieden, ob es sich um das Endzeichen handelt oder nicht. Diese Art der Namensausgabe sieht wie folgt aus:

```
BSOUT       =       $FFD2
;
            LDX     #0              ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE    LDA     NAME,X          ;ZEICHEN AUS DER TABELLE LESEN
            BEQ     ENDE            ;DAS ENDEZEICHEN ?,JA =>
            JSR     BSOUT           ;DAS ZEICHEN AUSGEBEN
            INX                     ;DEN INDEX ERHOEHEN
            JMP     SCHLEIFE        ;ZUM SCHLEIFENANFANG SPRINGEN
;
ENDE        RTS                     ;DAS PROGRAMM WIEDER VERLASSEN
NAME        .TEXT "SYBEX",0         ;NAMENSTABELLE
```

Bei dieser Programmart ist zu beachten, daß mit LDA NAME,X nur Tabellen mit einer Länge von 256 Bytes ausgelesen werden können. Sollte der Name länger und damit auch das Endzeichen weiter entfernt sein, findet das Programm dieses nicht mehr. Hat das Indexregister den Maximalwert von 255 erreicht, beginnt der Zähler wieder bei 0.

Eine beispielsweise 3000 Bytes entfernte Endmarke wird nie erreicht. Das Programm hört nicht mehr auf, Zeichen auszugeben. Die Schleife gehört, im Gegensatz zu den bisherigen Beispielen, zur Kategorie WHILE...DO...

Der Schleifenrumpf wird so lange durchlaufen, bis eine Bedingung nicht mehr erfüllt ist. Bei der Namensausgabe wird die Schleife durchlaufen, solange das Endzeichen nicht erreicht ist. Hier eine Gegenüberstellung der beiden Schleifenarten im Flußdiagramm und in Programmform:

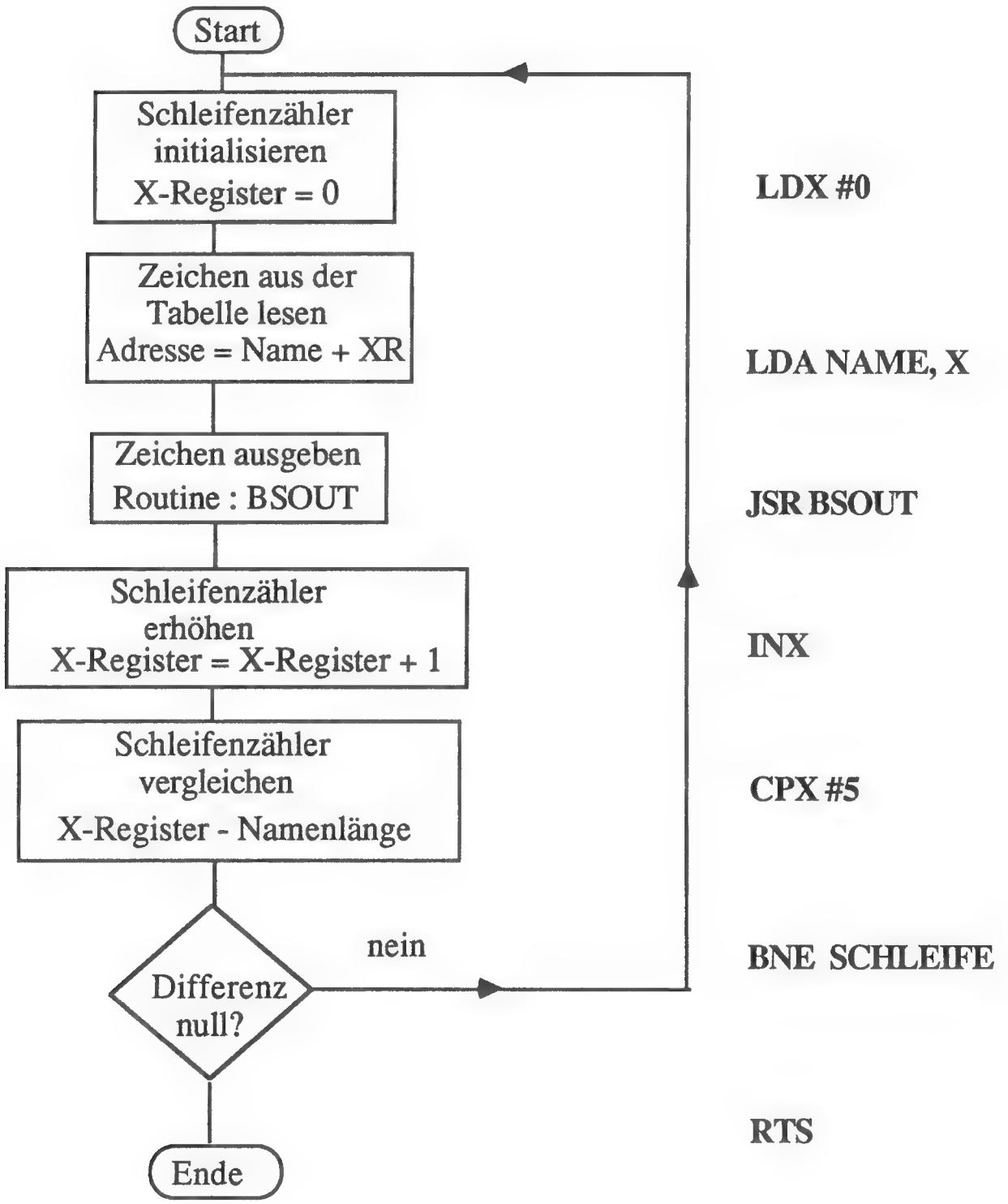


Abb. 4.1: BNE-Schleife

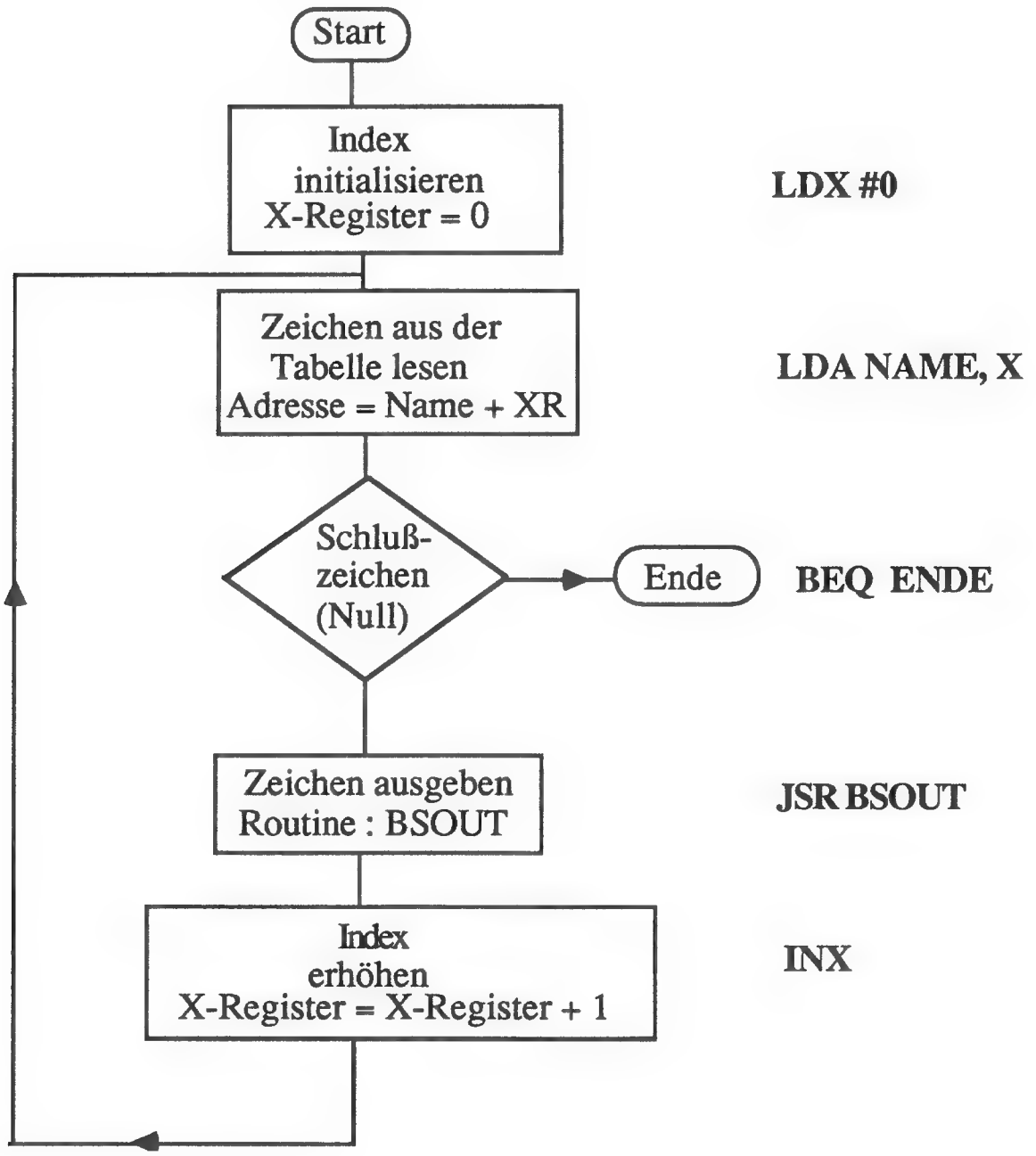


Abb. 4.2: JMP-Schleife

Aufgabe 4.8 Das letzte Ausgabeprogramm soll jetzt ihre gesamte Adresse ausgeben. Name, Straße und Ort sollen auf einzelnen Zeilen erscheinen. Das Ganze läßt sich durch eine geschickte Wahl des Ausgabetextes erreichen.

Aufgabe 4.9 Die Ausgabe des Namens über die verwendete WHILE-Schleife hat den beschriebenen Nachteil, daß bei einem zu langen Namen das Endzeichen nicht gefunden wird und das Programm nicht mehr aufhört, Zeichen auszugeben. Bauen Sie nach dem INX-Befehl eine Abfrage ein, die nach 200 Zeichen die Ausgabeschleife abbricht und damit das Problem löst.

Als Abschluß des Kapitels einige Beispiel für die Vergleichsbefehle. Bei der Namensausgabe sollen alle Vokale aussortiert und nicht auf dem Bildschirm

dargestellt werden. Nach dem Einlesen eines Zeichens wird dieses mit CMP-Befehl überprüft. Da dieser Befehl den Inhalt des Akkumulators nicht verändert, können die einzelnen Vergleiche mit den Vokalen hintereinander erfolgen.

```
BSOUT      =     $FFD2
           LDX   #0              ;INDEX = 0
SCHLEIFE   LDA   NAME,X          ;ZEICHEN AUS DER TABELLE LESEN
           BEQ   ENDE            ;DAS ENDEZEICHEN ?,JA =>
           CMP   #"A"            ;VOKAL "A" ?
           BEQ   WEITER          ;JA => (KEINE AUSGABE)
           CMP   #"E"            ;VOKAL "E" ?
           BEQ   WEITER          ;JA => (KEINE AUSGABE)
           CMP   #"I"            ;VOKAL "I" ?
           BEQ   WEITER          ;JA => (KEINE AUSGABE)
           CMP   #"O"            ;VOKAL "O" ?
           BEQ   WEITER          ;JA => (KEINE AUSGABE)
           CMP   #"U"            ;VOKAL "U" ?
           BEQ   WEITER          ;JA => (KEINE AUSGABE)
           JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
WEITER     INX                   ;DEN INDEX ERHOEHEN
           JMP   SCHLEIFE        ;ZUM SCHLEIFENANFANG SPRINGEN
;
ENDE       RTS                   ;DAS PROGRAMM WIEDER VERLASSEN
NAME       .TEXT "SYBEX",0       ;NAMENSTABELLE
```

Nach dem Einlesen des Zeichens wird, wie bereits bekannt, entschieden, ob es sich um das Endzeichen handelt. Danach wird mit dem CMP-Befehl festgestellt, ob es sich um einen der Vokale a, e, i, o oder u handelt. Stellt einer der Vergleichsbefehle eine Übereinstimmung fest, wird die Ausgabe (JSR BSOUT) übersprungen, und der entsprechende Vokal erscheint nicht.

Handelt es sich bei den Zeichen um einen Konsonanten,"fällt" dieser sozusagen durch alle Abfragen hindurch und trifft auf die Ausgabe. Auch hier dient wieder ein Flußdiagramm auf der nächsten Seite zur Veranschaulichung des Programmablaufs.

Mit diesem Beispiel besitzen Sie ein erstes Assemblerprogramm, das Datenverarbeitung betreibt. Die Daten, also Ihr Name, wird untersucht und in neuer Form, nämlich ohne Vokale, wieder ausgegeben. Das letzte Beispiel war sicher schwierig. Versuchen Sie sich aber auch an folgender Aufgabe.

Aufgabe 4.10 Das Programm soll jetzt statt der Vokale die Konsonanten verschlucken. Suchen Sie eine Lösung, ohne jeden einzelnen Konsonanten zu überprüfen. Dies wären ja immerhin 21 CMP- und BEQ-Befehle! Ein kleiner Tip: Sollen die Konsonanten verschluckt werden, dürfen nur Vokale ausgegeben werden (Bedingung umdrehen).

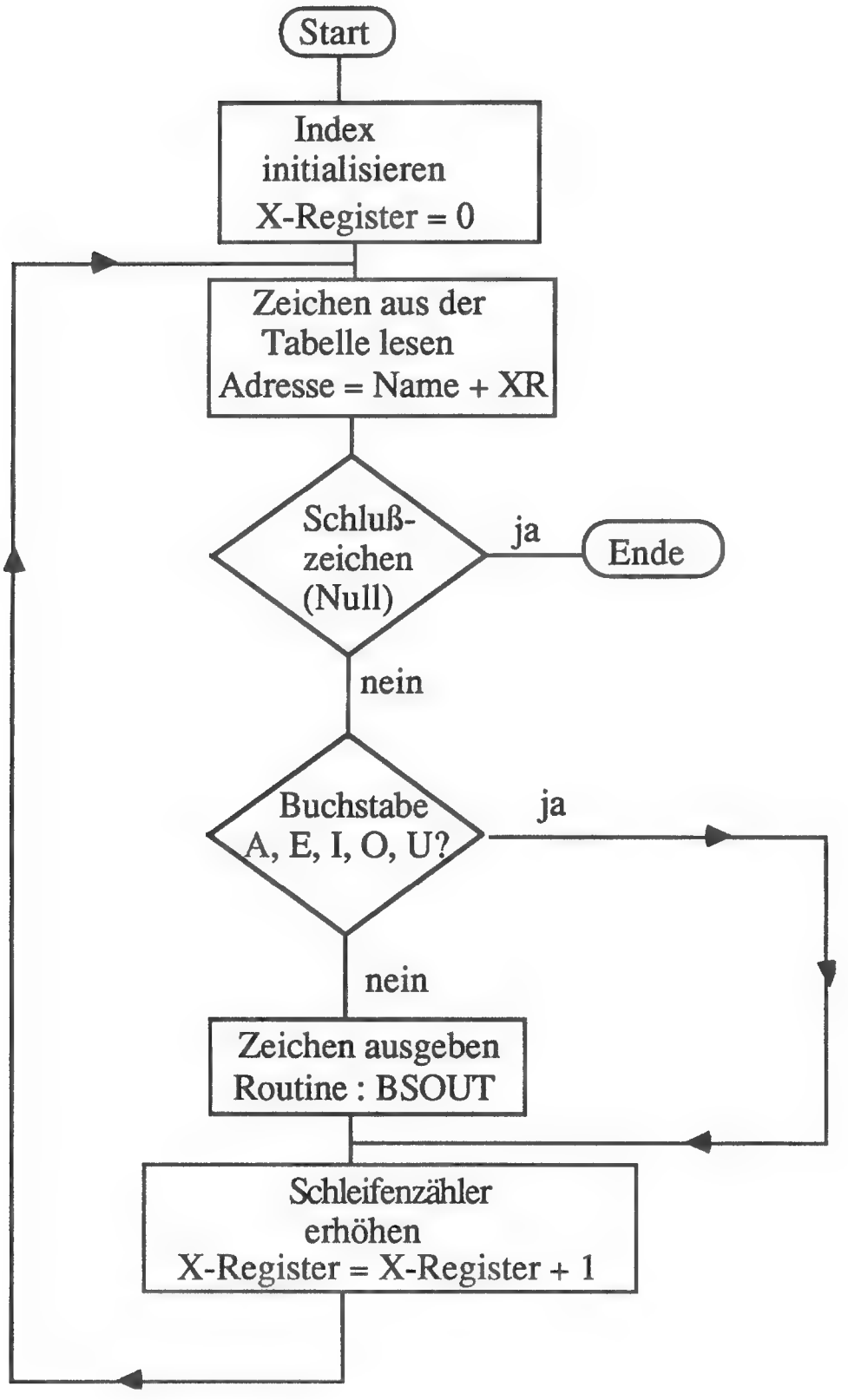


Abb. 4.3: Programmablauf

Aufgabe 4.11 Sie haben jetzt einige Erfahrung mit dem CMP-Befehl. Schreiben Sie ein Programm, das einen Text, begrenzt durch ein Endzeichen, ausgibt. Als Endzeichen verwenden Sie bitte ein Komma

Kapitel 5

Adressierungsarten

Wenn Sie die Befehle betrachten, die Sie in den letzten Kapiteln kennengelernt haben, werden Sie feststellen, daß ein und dasselbe Befehlswort sich mit verschiedenen folgenden Operanden, die Daten unterschiedlich aus dem Speicher holt.

Tatsächlich ist dieser getrennte Befehlsaufbau in der Assemblersprache bedeutend. Im Befehlswort, dem Operationcode, wird gespeichert, was gemacht werden soll, und in der nachfolgenden Angabe, dem Operanden, wird die Quelle bzw. das Ziel der Daten angegeben. Da mit diesen Daten eine Speicherzelle ausgewählt wird, wird dieser Befehlsteil auch Adreßteil genannt, und die verschiedenen Möglichkeiten heißen Adressierungsarten oder kurz Adressierarten.

Bisher haben Sie drei verschiedene Adressierarten kennengelernt. Die erste holt sich einen Zahlenwert direkt aus dem Programm, z.B. liest LDA #10 den Wert 10 in den Akkumulator.

Die zweite liest den Wert aus einer Speicherzelle. Im Befehl wird die Adresse dieser angegeben. STY 5000 z.B. überträgt den Inhalt des Y-Registers in die Speicherzelle 5000.

Die letzte Adressierart war auch die bisher komplizierteste. Hier wird die Adresse der Speicherzelle aus der Summe der im Operanden angegebenen Zahl und dem Inhalt des X- bzw. Y-Registers gebildet. Der Befehl LDA 4500,X nimmt als Speicheradresse für den Ladebefehl die Summe aus 4500 und dem Inhalt des X-Registers. Enthält dieses z.B. 85, wird die Speicherzelle 4585 adressiert.

Eine vierte Adressierart, die als solche nicht sofort erkannt wird, haben Sie auch schon gelernt. Befehle wie TAY, INX oder RTS teilen dem Prozessor bereits mit, von welchen Orten die Daten stammen müssen. Der TAY-Befehl z.B. sagt, daß die Daten aus dem Akkumulator stammen und ins Y-Register geleitet werden sollen. Diese ersten Adressierarten sind auch Vertreter ganz unterschiedlicher Kategorien. Es gibt solche, die eine Speicherzelle direkt ansprechen (LDA 5000, STX 1024), andere wiederum, die einen Index ver-

wenden und so auf ganze Bereiche des Speichers zugreifen. Eine weitere Gruppe benötigt überhaupt keine Angaben, sondern besitzt festgelegte Datenquellen und -ziele.

Bevor jetzt alle Adressierarten im einzelnen beschrieben werden, noch ein paar allgemeine Dinge, die für das Verständnis der Erklärungen notwendig sind. Eine Adresse umfaßt beim 6502- bzw. 8502-Prozessor immer 16 Bit. Aus Kapitel 3 wissen Sie bereits, daß 16 Bit durch 2 Bytes dargestellt werden.

Die 2 Bytes bekommen Namen. Das Byte mit den niederwertigen Zahlen (Bit 0-7) heißt entsprechend LO-Byte (LOw = niedrig) und das zweite Byte mit den höherwertigen Stellen (Bit 8-15) HI-Byte (HIgh = hoch). Im HI-Byte hat die unterste Stelle den Wert 256. Die gesamte dargestellte Zahl berechnet sich deshalb aus beiden Werten wie folgt:

(LO-Byte) + 256 * (HI-Byte) = Adresse

Hier ein paar Beispiele:

```
 10  +    256  *  20    =  5130
  0  +      4  * 256    =  1024
255  +    255  * 256    = 65535
```

Das letzte Beispiel ist auch gleichzeitig die maximal mit 16 Bits darstellbare Zahl und damit die größte erreichbare Adresse des 8502. Die Umrechnung einer Adresse in LO- und HI-Byte gestaltet sich etwas schwieriger. Zuerst wird der Zahlenwert durch 256 geteilt. Die Stellen vor dem Komma ergeben das HI-Byte.

Anschließend wird das HI-Byte mit 256 multipliziert und die Differenz mit dem Adreßwert gebildet. Als Ergebnis erhält man den Wert des LO-Bytes. Auch hier wieder ein paar Beispiele:

```
47602/256                 = 185.945... => HI-Byte  =  185
47602 – 256*185           = 242        => LO-Byte  =  242
55296/256                 = 216        => HI-Byte  =  216
da es keinen Rest gibt                 => LO-Byte  =    0
```

Bei diesen Berechnungen unterstützt der Assembler den Programmierer wieder. In mathematischen Berechnungen wird mit besonderen Zeichen vom nachfolgenden Ausdruck das LO-Byte bzw. HI-Byte berechnet. Das Kleinerzeichen (<) berechnet das LO-Byte (LO => kleiner) und das Größerzeichen (>) das HI-Byte (HI => größer). Hier wieder Beispiele, berechnet mit dem Befehl != von EDASS:

```
!=>47602                    !=<47602
00185                       00242

!=>55296                    !=<55296
00216                       00000

!$=>$ABCD                   !$=<$ABCD
    $00AB                   $00CD

!=<512+1                    !=>512+1
00001                       00002
```

Im letzten Beispiel wird das LO- bzw. HI-Byte von einem mathematischen Ausdruck gebildet. Klammern sind hier nicht erforderlich, da die Operatoren die Berechnung am Wert des gesamten nachfolgenden Ausdrucks vornehmen.

Für LO- und HI-Byte gibt es noch eine andere Interpretation. Der gesamte Speicherbereich des Prozessors wird dazu in 256 Seiten (englisch: page) eingeteilt. Die Seite wird mit dem HI-Byte ausgewählt und das Byte innerhalb der Seite mit dem LO-Byte. Die Interpretation des ersten Beispiels ergäbe dann: Seite 185, Byte 242.

Liest der Prozessor eine Adresse, muß diese in der Reihenfolge LO-Byte und dann HI-Byte im Speicher stehen. Im Operanden wird ihm nur die Position des LO-Bytes genannt. Der Prozessor weiß dann automatisch, daß das HI-Byte in der nachfolgenden Adresse abgelegt ist.

Soll also z.B. in den Speicherzellen 5000 und 5001 die Adresse 47602 stehen, muß die Speicherzelle 5000 den Wert 242 (LO-Byte) enthalten und 5001 den Wert 185 (HI-Byte). Dem Prozessor wird im Operanden die Adresse 5000 mitgeteilt. Genug der Vorreden. Wenden wir uns wieder den Befehlen des Prozessors zu. Ein Maschinenbefehl, also der übersetzte Assemblerbefehl, ist ähnlich wie ein Assemblerbefehl gegliedert.

Das erste Byte enthält einen Zahlenwert für das Befehlswort und eine Angabe, um welche Art von Adressierung es sich handelt. Ist eine Adresse notwendig, enthalten das nächste oder die beiden nächsten Bytes den Zahlenwert der Adresse. Beim Befehl STA 47602,Y stehen z.B. im Speicher die drei Bytes 153, 242 und 185. 153 sagt dem Prozessor, daß es sich um einen STA-Befehl mit sogenannter "absolut-indizierter" Adressierung handelt.

Das zweite und dritte Byte enthält, wie Sie sicher schon gemerkt haben, das LO- und HI-Byte der Adresse, zuerst das LO- und dann das HI-Byte. Besitzt ein Befehl keine Adresse, ist nur das Befehlsbyte erforderlich, womit der gesamte Befehl 1 Byte lang wäre.

Mit einer Adresse kann der Befehl 2 Bytes oder im Maximalfall 3 Bytes lang sein (1 Byte Befehlswort, 2 Bytes 16-Bit-Adresse). Die Länge eines Befehls hängt also nur von der verwendeten Adressierungsart ab. Jetzt zur Beschreibung der verschiedenen Adressierungsarten. Es wird das Eingabeformat als Anwendungsbeispiel in der Überschrift, das Funktionsprinzip, die sich ergebende Befehlslänge, die Befehle, in denen sie vorkommen, und ein kurzes Verwendungsbeispiel angegeben. Bei Adressierarten, die indiziert arbeiten, wird als Beispiel das aus dem letzten Kapitel bekannte Problem der Textaus- gabe verwendet.

Implizite oder inhärente Adressierung – INX

Bei dieser Art von Adressierung wird keine Adresse benötigt. Der Prozessor weiß aus dem Befehlswort, wo Quelle und Ziel der Daten liegen. Befehle, die diese Adressierart verwenden, sind deshalb nur ein Byte lang. Die Befehle, die diese Adressierungsart verwenden, lassen sich in zwei Gruppen einteilen. Bei der ersten Gruppe werden Daten innerhalb des Prozessors verändert:

CLC, CLD, CLI, CLV, DEX, INX, INY, NOP, TAX, TAY, TSX, TXA, TXS, TYA, SEC, SED, SEI

Bei der zweiten Gruppe werden zusätzlich Speicherinhalte beeinflußt:

BRK, PHA, PHP, PLA, PLP, RTI, RTS

Eine dritte Gruppe von Befehlen kann auch hinzugenommen werden. Eigentlich handelt es sich dabei um Befehle mit Akkumulator-Adressierung, d.h. der Inhalt des Akkumulators wird durch den Befehl angesprochen. Im Assemblerprogramm wird üblicherweise als Adresse ein "A" eingegeben. Da dieses jedoch bei vielen Assemblern, so auch bei EDASS, entfallen kann und die Befehle auch nur ein Byte lang sind, wird diese Adressierungsart in diese Gruppe eingeordnet. Hier die entsprechenden Befehle:

ASL, LSR, ROL, ROR

Beispiele für Befehle mit impliziter Adressierung werden im folgenden noch zahlreich erscheinen.

Unmittelbare Adressierung – LDX #8-Bit-Zahlenwert

Diese Adressierungsart haben Sie bereits ausführlich kennengelernt. Der Prozessor liest als Datenquelle einen Wert direkt aus dem Programm. Da als

Werte nur 8-Bit-Zahlen zugelassen sind (0 bis 255), ist der gesamte Befehl zwei Byte lang, ein Byte für das Befehlswort und eins für den Zahlenwert. Verwendet wird diese Adressierart zum Laden von Registern, zur Arithmetik und zum Vergleichen der Registerinhalte. Im einzelnen lauten die Befehle:

ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC

Hier ein Beispiel, das den Inhalt der Speicherzelle 5000 überprüft und einen Stern ausgibt, falls sich dort der Wert 123 befindet. Geben Sie das Programm ein, und verändern Sie den Inhalt der Speicherzelle 5000 mit dem BASIC-Befehl POKE.

```
BSOUT     =     $FFD2
          LDA   5000        ;AKKU. = SPEICHERZELLE 5000
          CMP   #123        ;UERPRUEFEN
          BNE   ENDE        ;WERT 123 ?,NEIN =>
          LDA   #"*"        ;AKKU. = ASCII-CODE EINES STERNS
          JSR   BSOUT       ;DEN STERN AUSGEBEN
ENDE      RTS               ;DAS PROGRAMM WIEDER VERLASSEN
```

Absolute Adressierung – LDA 16-Bit-Wert

Auch diese Adressierart ist Ihnen bereits bekannt. Im Befehl wird die Adresse einer Speicherzelle direkt vorgegeben. Diese Adresse wird in zwei Bytes gespeichert, womit ein Befehl eine Länge von drei Bytes bekommt. Dies ist sicher eine der einfachsten Adressierungsarten, da sie immer direkt mit der angesprochenen Adresse korrespondiert. Von allen Befehlen, die nicht über implizite Adressierung verfügen, wird diese Adressierart verwendet. Hier wieder eine Liste:

ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY

Beispiele für diese Adressierung werden Sie bei allen folgenden Beschreibungen zur Genüge finden.

Zero-Page-Adressierung – LDA 8-Bit-Wert

Bei dieser Adressierart handelt es sich um eine Teilgruppe der absoluten Adressierung. Als Unterschied sind nämlich nur 8-Bit-Werte, also Zahlen von 0 bis 255, erlaubt. Innerhalb der möglichen 65536 Speicherzellen liegen die damit gewählten Zellen in den ersten 256 Bytes. Das HI-Byte einer solchen Adresse ist immer null. Wenn Sie sich jetzt an das oben zu LO- und HI-Byte

Erklärte erinnern, wissen Sie, daß es sich dabei um die Seite null aus dem Speicher handelt. Damit ist auch der Ursprung des Namens klar (Zero = Null, Page = Seite). Welchen Vorteil hat diese Adressierung? Erstens braucht sie weniger Speicher, da nur ein Byte für die Adresse, und für den Befehl insgesamt zwei Bytes benötigt werden. Aber diese Adressierart ist auch schneller. Ein HI-Byte muß nicht eingelesen werden, sondern der Prozessor setzt es selbst hinzu. Diese Vorteile sind, wenn es Ihnen jetzt auch nicht so erscheint, sehr bedeutend. Sie sollten immer versuchen, Zähler und andere häufig benutzte Speicherzellen in die Zero-Page zu legen. Dies gestaltet sich etwas schwierig, da BASIC die Zero-Page komplett belegt. In Anhang G finden Sie aber eine Tabelle aller Speicherzellen, die kurzzeitig verwendet werden dürfen.

Ein Punkt ist noch nicht geklärt: Wie wird EDASS diese Adressierart mitgeteilt, das Eingabeformat ist ja identisch mit der absoluten Adressierung? Der Assembler wählt hier immer die bestmögliche Adressierung aus. Wird als Adreßwert eine Zahl kleiner als 256 eingegeben, nimmt der Assembler Zero-Page-Adressierung für den Objektcode und bei größeren Werten absolute Adressierung. Jetzt ist auch klar, daß die gleichen Befehl wie bei der absoluten Adressierung eine Zero-Page-Adresse zulassen. Als Beispiel einmal kein Programm zum Eintippen, sondern ein Assemblerlisting, das die Vorteile der Adressierungsart zeigt.

```
F1300  A5 80     0001     LDA   128
F1302  AD 00 04  0002     STA   1024
F1305  AE 00 D8  0003     LDX   55296
F1308  86 41     0004     STX   65
```

Indizierte Adresse – CMP 8/16-Bit-Wert,X – CMP 8/16-Bit-Wert,Y

Bei dieser Adressierungsart wird zum Operanden der Inhalt des X- bzw. Y-Registers addiert. Sie dient vor allem dem Zugriff auf Tabellen. Als Zahl im Operanden wird die Basisadresse der Tabelle eingegeben, und das gewählte Register zeigt auf einen Wert in der Tabelle. Das X- und das Y-Register werden wegen dieser Funktion auch Index-Register genannt.

Beide Register können Werte bis 255 aufnehmen, folglich können mit dieser Adressierungsart Tabellen mit 256 Elementen angesprochen werden. In bezug auf die Länge der verwendeten Zahl und damit die Befehlslänge gibt es wieder zwei Versionen der Adressierart, eine 16-Bit-Adresse und eine 8-Bit-Adresse, bei der wieder auf die Zero-Page zugegriffen wird.

Auch wählt der Assembler jeweils den günstigeren Maschinenbefehl aus. Die Verwendung des X- bzw. Y-Registers ist nicht austauschbar. Auch erlaubt

nicht jeder Befehl eine 8-Bit-Adressierung. Aber hier hilft wieder der Assembler. Er wählt nur eine 8-Bit-Adressierung aus, wenn dies auch möglich ist.

Die erste Befehlsliste zeigt alle Befehle, die das Index-Register X verwenden und 8- und 16-Bit-Adressen erlauben:

ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA

Einzige Ausnahme in der Gruppe der mit dem X-Register indizierten Befehle bildet STY. Bei diesem Befehl kann nur eine 8-Bit-Adresse verwendet werden. Die nächste Liste enthält alle Befehle die mit dem Y-Register indiziert werden und über eine 16-Bit-Adressierung verfügen:

ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA

Als einziger Befehl in dieser Reihe erlaubt der LDX zusätzlich eine 8-Bit-Adresse. Der STX-Befehl erlaubt wieder nur eine 8-Bit-Adresse. An diesen Unterschieden wird deutlich, daß X- und Y-Register nicht austauschbar sind. An zwei weiteren Adressierarten wird dies noch viel augenscheinlicher. Als Beispiel wird ein 12 Zeichen langer Text ausgegeben.

```
BSOUT     =    $FFD2
          LDY  #0                  ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE  LDA  TEXT,Y              ;ZEICHEN AUS DEM TEXT LESEN
          JSR  BSOUT               ;DAS ZEICHEN AUSGEBEN
          INY                      ;DEN SCHLEIFENZAEHLER ERHOEHEN
          CPY  #12                 ;DAS ZAEHLERENDE ERREICHT ?
          BNE  SCHLEIFE            ;NEIN =>
          RTS                      ;DAS PROGRAMM WIEDER VERLASSEN
;
TEXT      .TEXT "SYBEX-VERLAG"
```

Nach-indizierte indirekte Adressierung – LDA (8-Bit-Wert),Y

Diese Adressierung ist wohl die schwierigste, aber auch eine der wichtigsten bei der Programmerstellung. Das Wort "indirekt" im Namen besagt, daß im Operanden nicht die Adresse der gewünschten Speicherzelle steht, sondern nur die Position, an der diese abgelegt ist. Also weiß der Prozessor z.B. beim Wert 55, daß die eigentliche Adresse in der Speicherzelle 55 zu finden ist.

Da aber zwei Bytes für eine komplette Adreßangabe notwendig sind, liest er aus der Speicherposition 55 das LO-Byte und aus 56 das HI-Byte. Übrigens, die Speicherposition der Adresse darf nur in der Zero-Page liegen, also in den

ersten 256 Bytes. Ein Befehl mit dieser Adressierung ist deshalb auch nur zwei Bytes lang. Der Begriff "indirekt" wäre geklärt. Mit indiziert wird wieder ausgedrückt, daß der Inhalt eines Registers, hier immer des Y-Registers, zur Adresse addiert wird. Nachdem der Prozessor die komplette 16-Bit-Adresse aus dem Speicher gelesen hat, wird der Inhalt des Y-Registers zu dieser addiert. Jetzt wird auch klar, warum es nach-indiziert heißt: Es wird ja zuerst die Adresse aus dem Speicher gelesen und dann der Index addiert.

Anwendung findet der Befehl immer beim Zugriff auf Tabellen, die größer als 256 Bytes sind. Die in der Zero-Page stehende Adresse dient als Zeiger in die Tabelle. Das Y-Register übernimmt die Aufgabe eines feineren Zeigers.

Soll z.B. ein im Computer befindliches BASIC-Programm ausgegeben werden, kann der Adreßzeiger auf den Zeilenanfang weisen, und das Y-Register spricht jedes einzelne Zeichen der Zeile an. Je nach Wert des Registers wird relativ zum Zeiger, also dem Zeilenanfang, das erste Byte, zweite Byte usw. ausgelesen.

Noch ein weiteres Beispiel: Wird z.B. ein Speicherbereich kopiert, zeigt ein Zeiger aus der Zero-Page auf den Quellbereich und ein zweiter auf den Zielbereich. Während die einzelnen Bytes übertragen werden, werden beide Zeiger erhöht. Die folgenden Befehle erlauben diese Adressierung:

ADC, AND, CMP, EOR, LDA, ORA, SBC, STA

Als Beispiel wird wieder ein zwölf Zeichen langer Text ausgegeben.

```
BSOUT     =      $FFD2
ZEIGER    =      250
          LDA    #<TEXT          ;LO-BYTE DER ADRESSE SPEICHERN
          STA    ZEIGER
          LDA    #>TEXT          ;HI-BYTE DER ADRESSE SPEICHERN
          STA    ZEIGER+
          LDY    #0              ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE  LDA    (ZEIGER),Y      ;ZEICHEN AUS DEM TEXT LESEN
          JSR    BSOUT           ;DAS ZEICHEN AUSGEBEN
          INY                    ;DEN SCHLEIFENZÄHLER ERHOEHEN
          CPY    #12             ;DAS ZAEHLERENDE ERREICHT?
          BNE    SCHLEIFE        ;NEIN =>
          RTS                    ;DAS PROGRAMM WIEDER VERLASSEN
;
TEXT              .TEXT "SYBEX-VERLAG"
```

Als Zeiger dienen die Speicherzellen 250 und 251. Mit den beiden Arithmetikzeichen "<" und ">" wird das LO- und HI-Byte des Textanfangs beim Assemblieren errechnet und in den Objektcode eingebaut. Das LO-Byte wird in

Speicherzelle 250 geschrieben. Die Adresse der Speicherzelle 251, zur Aufnahme des HI-Bytes, wird aus dem Label ZEIGER errechnet. Dies ist eine oft verwendete Methode in Assemblerprogrammen, um die Position des HI-Bytes zu bestimmen.

Wird die Adresse des Zeigers verändert, muß nur der Wert eines Labels korrigiert werden. Wäre das LO- und HI-Byte in zwei getrennten Labels abgelegt, müßten beide neu bestimmt werden. Ein Fehler hierbei kann gravierende Folgen haben. Man stelle sich nur vor, LO- und HI-Byte wären auf einmal vertauscht.

Vor-indizierte indirekte Adressierung – LDA (8-Bit-Wert,X)

Diese Adressierung kann wieder aus dem Namen erschlossen werden. "Indirekt" bedeutet, daß die Adresse der gewünschten Speicherzelle nicht direkt im Programm eingegeben wird, sondern dort nur auf eine Speicherzelle verweist, die deren Wert enthält. Vor-indiziert besagt jetzt, daß zu dem im Befehl eingegebenen Wert erst der Inhalt eines Registers, hier immer das X-Register, addiert wird und erst dann aus der dabei entstehenden Speicherposition der Wert der Adresse entnommen wird.

Die errechnete Position muß wieder in den ersten 256 Bytes liegen, weshalb ein entsprechender Befehl immer zwei Bytes lang ist. Enthält z.B. das X-Register den Wert 14 und bearbeitet der Prozessor den Befehl STA (90,X), addiert er zu 90 den Inhalt des X-Registers, also 14, und erhält mit 104 die Position der eigentlichen Adresse.

Aus der Speicherzelle 104 wird das LO-Byte und aus 105 das HI-Byte gelesen. Anwendung findet der Befehl sehr selten. Folgender Fall wäre denkbar: Es muß z.B. auf fünf Tabellen in schnellem Wechsel zugegriffen werden. In zehn Speicherzellen (pro Zeiger zwei Bytes) der Zero-Page werden die Adressen der fünf Tabellenanfänge abgelegt. Das Programm wählt dann die benötigte Tabelle mit dem X-Register aus. Die gleichen Befehle, die nach-indizierte indirekte Adressierung zulassen, kennen auch diese Adressierung:

ADC, AND, CMP, EOR, LDA, ORA, SBC, STA

Indirekte Adressierung – JMP (16-Bit-Wert)

Wenn Sie die beiden letzten Adressierungen verstanden haben, müßte Ihnen die Funktionsweise dieser Anwendungsart bereits aus dem Namen klar sein. Es wird hier indirekt auf eine Adresse zugegriffen. Die im Befehl angegebene

Position der Adresse darf hier jedoch 16 Bit umfassen, womit sie im gesamten Speicher liegen darf. Die Befehlslänge beträgt deshalb auch drei Bytes. Der Befehl JMP erlaubt als einziger diese Adressierung. Der Prozessor nimmt den Wert aus dem Operanden, liest aus der damit bestimmten Speicherzelle das LO-Byte und aus der nachfolgenden das HI-Byte der Adresse.

Zu dieser Stelle führt der Prozessor dann einen Sprung aus. Um derartige Sprungadressen in den Objektcode einzubauen, besitzt der Assembler die Anweisung .WORD. Diese fügt wie der .BYTE-Befehl Zahlenwerte in das Maschinenprogramm ein.

Die Werte dürfen jetzt aber 16 Bit lang sein. Die Werte werden in der Reihenfolge LO-Byte, HI-Byte, also der vom Prozessor verlangten Reihenfolge für Adressen, gespeichert. Natürlich können Sie den Befehl auch für andere Zwecke als Sprungadressen einsetzen

```
BSOUT     =        $FFD2
          LDA      #"%"          ;ASCII-CODE VON "%" HOLEN
          JMP      (SPRUNG)      ;ZEICHEN AUSGEBEN
;
SPRUNG    .WORD    BSOUT         ;SPRUNGADRESSE
```

Zur Verdeutlichung das Assemblerlisting:

```
                  0003 BSOUT =       $FFD2
F1300 A9 25       0005       LDA     #"%"          ;ASCII-CODE...
F1302 6C 05 13    0006       JMP     (SPRUNG)      ;ZEICHEN AU...
                  0007;
F1305 D2 FF       0008       .WORD   BSOUT         ;SPRUNGADRE...
```

Sicher werden Sie das sonst gewohnte RTS im Programm vermissen. Aber überlegen wir einmal: Am Ende der Routine BSOUT befindet sich auch ein RTS, sonst könnte der Prozessor nicht ins aufrufende Programm zurückkehren.

Warum soll dieser RTS-Befehl nicht gleich zum Verlassen des Beispielprogramms dienen? Dies ist auch kürzer. Die Routine BSOUT darf für diesen Effekt natürlich nicht mehr mit JSR aufgerufen werden, sondern mit JMP. In Assemblerprogrammen, so auch in den ROMs des C128, ist oft am Ende eines Unterprogramms ein JMP-Befehl zu finden.

Beim Verlassen des Unterprogramms wird noch schnell ein anderes Unterprogramm aufgerufen. Das genaue Verständnis der verschiedenen Adressierungsarten ist sehr wichtig, denn sie erwecken ein Programm eigentlich erst zum Leben. Die Adressierungsarten sind sozusagen die "Arme" der Befehle,

mit denen sie in den Speicher greifen, Daten holen und wieder ablegen. Das Befehlswort selbst stellt nur die Funktion dar, die Adressierarten liefern die Daten.

Bei der Beschreibung der Adressierarten ist Ihnen sicher aufgefallen, daß nicht alle Befehle über das volle Spektrum der Adressierungen verfügen. Bei Ihren ersten eigenen Programmen wird es hier sicher einige Fehlgriffe geben, aber der Assembler weist Sie auf jede unzulässige Kombination von Befehlswort und Adressierungsart mit folgender Fehlermeldung hin:

```
?UNERLAUBTE BEFEHLSKOMBINATION
```

Aufgabe 5.1 Bauen Sie das Programm zur Ausgabe eines Textes mit Endmarke aus dem letzten Kapitel mit nach-indizierter Adressierung auf. Als Speicherzelle in den ersten 256 Bytes stehen Ihnen die Adresse 250 bis 254 zur Verfügung.

Aufgabe 5.2 Jetzt eine etwas schwierigere Aufgabe. Das Ergebnis ist zwar nicht besonders sinnvoll, aber doch nett anzuschauen. Zwei Texte sollen zusammen ausgegeben werden. Die einzelnen Zeichen sollen abwechselnd erscheinen. Erst eins vom ersten Text, dann eins vom zweiten und dann wieder vom ersten usw. Das Programm soll abbrechen, wenn in einem der beiden Texte die Endmarke erreicht ist. Als Hilfe können Sie ja ein bisheriges Ausgabeprogramm als Grundlage für eine Erweiterung nehmen.

Kapitel 6

Eins mal eins, Arithmetik in Assemblerprogrammen

Vorzeichenbehaftete Zahlen

In den letzten Kapiteln sind Sie bereits öfters mit der Tatsache konfrontiert worden, daß der Prozessor negative Zahlen kennt. Nun soll dieses Rätsel gelöst werden. In Wirklichkeit gibt es in der Maschinensprache keine negativen Zahlen, sondern es wird einfach ein Teil der positiven Zahlen als negative interpretiert.

Die Zahlen von 0 bis 127 sind positiv und die Zahlen von 128 bis 255 negativ. Betrachten wir ein paar Beispiele in binärer Darstellung

```
 85      = 01010101
127      = 01111111
 47      = 00101111

128      = 10000000
169      = 10101001
255      = 11111111
```

Sie sehen, daß bei allen Zahlen die als negativ aufgefaßt werden das höchste Bit, also Bit 7, gesetzt ist.

Welchem tatsächlichen Wert entsprechen die Zahlen von 128 bis 255 aber jetzt? Man könnte denken, daß ein positiver Wert durch Setzen des siebten Bits in einen negativen verwandelt wird, also –1 der Zahl 129 (10000001) entspricht.

Mit dieser Festlegung könnte man aber auch eine Zahl wie –0 (10000000) darstellen, die gar nicht existiert.

Man hat deshalb eine andere Konvention definiert, die viele Vorteile bietet. Eine positive Zahl wird durch Bildung ihres Komplements (0 wird zu 1 und 1 wird zu 0) und Addition von 1 in eine negative Zahl verwandelt. Hierzu ein Beispiel:

```
                01001110    (78)
Komplement:     10110001   (177)
              +        1
                --------
                10110010   (178)
```

Das einfache Komplement wird auch Einerkomplement genannt, und das um eins erhöhte heißt Zweierkomplement. Die Arithmetik, die mit diesen Zahlen arbeitet, wird deshalb auch als Zweierkomplement-Arithmetik bezeichnet. Die Vorteile dieser Darstellung werden klar, wenn ein paar Zahlen in negative umgewandelt werden:

```
                00000000     (0)
Komplement:     11111111   (255)
              +        1
                --------
               (1)00000000     (0)


                00000001     (1)
Komplement:     11111110   (254)
              +        1
                --------
                11111111   (255)
```

Bei der Umwandlung bleibt die Zahl Null erhalten. Es gibt nicht +0 und –0.

Erinnern Sie sich jetzt einmal an den Effekt, der eintritt, wenn die Zahl 0 durch einen Dekrement-Befehl, z.B. DEX, vermindert wird. Es entsteht die Zahl 255. Streng mathematisch müßte –1 entstehen. Betrachten Sie das obige Beispiel, so stellen Sie fest, daß 255 die Zweierkomplement-Darstellung von –1 ist. Der umgekehrte Effekt klappt natürlich auch: 255, also –1, um 1 erhöht (inkrementiert), ergibt das mathematisch korrekte Ergebnis 0. Hieraus leitet sich auch eine ganz wichtige Eigenschaft ab: Negative Zahlen werden richtig dekrementiert und inkrementiert.

```
 -5   =>   251   -->  +1   -->   252   =>   -4
-68   =>   188   -->  +1   -->   189   =>  -67

 -7   =>   249   -->  -1   -->   248   =>   -8
-55   =>   201   -->  -1   -->   200   =>  -56
```

Eine weitere angenehme Eigenschaft haben diese Zahlen. Die Addition einer positiven und der dazugehörigen negativen Zahl ergibt null.

```
     0 0 1 1 1 0 1  0        (58)
   + 1 1 0 0 0 1 1  0        (198 = -58)
   ---------------------
(1)  0 0 0 0 0 0 0  0        (0)
```

Einen Ausschnitt der Zahlengeraden zeigt folgendes Bild. Es veranschaulicht nochmals alle Effekte und Vorteile der Zweierkomplement-Darstellung:

```
(127)    01111111  ::   127
                    ^^
(3)      00000011  ::   3
(2)      00000010  ::   2
(1)      00000001  ::   1
         00000000  ::   0
(255)    11111111  ::  - 1
(254)    11111110  ::  - 2
(253)    11111101  ::  - 3
                    ^^
(128)    10000000  ::  - 128
```

Die beschriebenen Zahlgrenzen für negative und positive Zahlen gelten nur für 8 Bit lange Zahlen. Liegen z.B. Binärzahlen mit einer Länge von 16 Bit vor, ergeben sich neue Grenzen. In diesem Fall werden die Zahlen von 0 bis 32767 als positiv und die Zahlen von 32768 bis 65535 als negativ interpretiert.

Auch hier gilt wieder die Regel, daß bei einer negativen Zahl das höchste Bit gesetzt ist. Bei einer 16-Bit-Zahl hat dieses jedoch den Stellenwert 32768.

Um die Zweierkomplementdarstellung einer negativen Zahl zu bilden, gibt es eine schnelle Methode:

```
  -1:    256  -   1   =   255
 -20:    256  -  20   =   236
  -1:  65536  -   1   = 65535
-123:  65536  - 123   = 65413
```

Der umgekehrte Weg ist natürlich auch möglich:

```
   236   -  256     =   - 20
 65413   -65536     =   -123
```

Kehren wir zurück zur Assemblerprogrammierung. Zwei Branch-Befehle überprüfen nach einer Operation das Vorzeichen des Ergebnisses.

BPL – Branch if PLus (N=0)
– Verzweige, wenn positiv

BMI – Branch if MInus (N=1)
– Verzweige, wenn negativ

Im Statusregister zeigt das N-Flag (negativ) an, ob ein Befehl ein negatives Ergebnis geliefert hat. Es kann sich dabei um einen Lade-, Arithmetik- oder sonstigen Befehl handeln.

Das Statusbit N korrespondiert immer mit Bit 7 des Endergebnisses. Wird z.B. mit dem Befehl LDA #180 der Akkumulator geladen, ist das Bit 7 gesetzt, und folglich zeigt auch das Statusflag einen negativen Wert an. Das Programm zur Ausgabe eines Textes verwendet jetzt einen negativen Zahlenwert als Endmarke. Hier das Programm:

```
BSOUT      =      $FFD2
           LDX    #0          ;INDEX=0
SCHLEIFE   LDA    TEXT,X      ;ZEICHEN AUS DEM TEXT LESEN
     BMI   ENDE               ;NEGATIV/ENDMARKE ?,JA =>
     JSR   BSOUT              ;ZEICHEN AUSGEBEN
     INX                      ;INDEX ERHOEHEN
     JMP   SCHLEIFE           ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                ;DAS PROGRAMM VERLASSEN
;
TEXT       .TEXT "SYBEX-VERLAG",128
```

Es soll jetzt nicht der Eindruck entstehen, daß alle Zahlenwerte nach der Zweierkomplement-Darstellung behandelt werden müssen. Dieses ist eine reine Interpretationsfrage. Wenn Sie nur positive Zahlen in Ihrem Programm wünschen, beachten Sie einfach das Negativ-Flag nicht. Rechnungen mit rein positiven Zahlen sind sogar oft notwendig. Bestimmt ein Programm z.B. die Anfangsadresse einer Tabelle oder einer Bildschirmzeile, würde man durch eine negative Interpretation der Ergebnisse dieses mißdeuten.

Nochmal addieren und subtrahieren

In diesem Abschnitt werden zwei Zahlen addiert bzw. subtrahiert unter Beachtung aller Regeln und Besonderheiten. Zur Einführung einmal die schriftliche Addition zweier Binärzahlen. Alle Übertrage werden mitnotiert:

```
 01011101   (93)
+11011010   (21)
11 11
 ---------
 00110111   (55)
```

Bei der Addition dieser zwei Zahlen ist ein Übertrag zu einem nicht vorhandenen Bit 8 entstanden. Dieser Übertrag müßte von einer weiteren Addition verarbeitet werden, was wie folgt aussehen würde:

```
  00000000 nachfolgende Stellen
+        1 Übertrag
  ---------
  00000001
```

Durch die Weitergabe des Übertrages können zwei 8-Bit-Additionen zu einer 16-Bit-Addition gekoppelt werden. Auch hier wieder ein Beispiel:

```
    00010010     11010001
+   01000010     10111011
       1 1 <- 1111   11
  ----------   ----------
    01010101     10001100
```

Wie wird ein Übertrag vom Prozessor behandelt? Das Auftreten eines solchen wird im Statusregister im Übertrags-Flag oder kurz C-Flag angezeigt. Man könnte auch sagen, der Übertrag wird im Statusflag gespeichert. Die Abkürzung "C" steht für das englische Wort Carry, was Übertrag heißt. Die Anzeige und Speicherung des Übertrags wäre sichergestellt. Die Weiterverarbeitung übernimmt der ADC-Befehl. Ausgeschrieben heißt dieser Befehl "Add with Carry", das heißt, addiere unter Einbeziehung eines Übertrags. Bei jeder Addition mit dem ADC-Befehl wird außer den zwei Datenbytes auch der Inhalt des Übertrag-Flags verarbeitet.

Bei einer einzelnen 8-Bit-Addition möchte man mit dem Flag, das durch die vorigen Operationen einen undefinierten Zustand hat, nicht aus Versehen einen Übertrag mitaddieren. Das Übertrags-Flag muß zuvor gelöscht werden.

CLC – CLear Carry
– Lösche den Übertrag

Wenn Sie sich an die ersten Beispiele zum ADC-Befehl erinnern, wird Ihnen klar, warum der Akkumulator mit dem CLC auf die Addition "vorbereitet" werden mußte. Damit ein korrektes Ergebnis entsteht, muß das C-Flag gelöscht werden. Gleich noch zwei weitere Befehle:

BCC – Branch if Carry Clear (C=0)
– Verzweige, wenn Übertrag gelöscht

BCS – Branch if Carry Set (C=1)
– Verzweige, wenn Übertrag gesetzt

Diese zwei Befehle führen einen relativen Sprung in Abhängigkeit vom Übertrags-Flag aus. Das Auftreten eines Übertrags bei der Addition kann so festgestellt werden und, z.B. durch eine Ausgabe, dem Benutzer mitgeteilt werden. Genau dies soll das folgende Programmbeispiel auch machen. Es werden zwei 8-Bit-Zahlen, die das Programm aus dem Speicher entnimmt, addiert.

Das Ergebnis wird ebenfalls im Speicher abgelegt. Zusätzlich wird durch eine Ausgabe angezeigt, ob ein Übertrag aufgetreten ist.

```
BSOUT   =    $FFD2
SUM1    =    250        ;SPEICHERZELLE DES 1. SUMMANDEN
SUM2    =    251        ;SPEICHERZELLE DES 2. SUMMANDEN
ERG     =    252        ;SPEICHERZELLE DES ERGEBNISSES
;
        LDA  SUM1       ;AKKU. MIT 1. SUMMANDEN LADEN
        CLC             ;DAS UEBERTRAGS-FLAG LOESCHEN
        ADC  SUM2       ;DEN 2. SUMMANDEN ADDIEREN
        STA  ERG        ;DAS ERGEBNIS SPEICHERN
        BCC  ENDE       ;EIN UEBERTRAG ?,NEIN =>
        LDA  #"C"       ;AKKU. = ASCII-CODE VON "C"
        JSR  BSOUT      ;DAS ZEICHEN AUSGEBEN
ENDE    RTS             ;DAS PROGRAMM WIEDER VERLASSEN
```

Geben Sie das Programm ein. In die Speicherzellen 250 und 251 schreiben Sie mit POKE die beiden Summanden. Starten Sie dann das Programm. Als Ausgabe erscheint jetzt eventuell ein "C", das Ihnen sagt, daß ein Übertrag aufgetreten ist. Das zahlenmäßige Endergebnis können Sie mit PEEK aus der Speicherzelle 252 auslesen. Hier ein paar Beispiele:

```
POKE 250,50:POKE 251,145
READY.
!GO $1300
AC: ...
READY.
?PEEK (252)
195
READY.

POKE 250,120:POKE 250,200
READY.
!GO $1300
C
AC: ...
READY.
?PEEK (252)
64
READY.
```

Jetzt sollen, wie bereits oben beschrieben, zwei 8-Bit-Additionen zu einer 16-Bit-Addition gekoppelt werden.

```
SUMLO1  =   250       ;LO-BYTE DES 1. SUMMANDEN
SUMHI1  =   251       ;HI-BYTE DES 1. SUMMANDEN
SUMLO2  =   252       ;LO-BYTE DES 2. SUMMANDEN
SUMHI2  =   253       ;HI-BYTE DES 2. SUMMANDEN
ERGLO   =   252       ;LO-BYTE DES ERGEBNISSES
ERGHI   =   253       ;HI-BYTE DES ERGEBNISSES
;
        LDA SUMLO1    ;LO-BYTE DES 1. SUMMANDEN HOLEN
        CLC           ;DAS UEBERTRAGS-FLAG LOESCHEN
        ADC SUMLO2    ;LO-BYTE DES 2. SUMM. ADDIEREN
        STA ERGLO     ;ENDERGEBNIS LO-BYTE SPEICHERN
        LDA SUMHI1    ;HI-BYTE DES 1. SUMMANDEN HOLEN
        ADC SUMHI2    ;HI-BYTE DES 2. SUMM. ADDIEREN
        STA ERGHI     ;ENDERGEBNIS HI-BYTE SPEICHERN
        RTS           ;DAS PROGRAMM WIEDER VERLASSEN
```

Die jetzt 16 Bit umfassenden Summanden müssen in zwei Speicherzellen übergeben werden. Wie bei 16-Bit-Adressen werden die Zahlen in LO- und HI-Byte geteilt. Entsprechend heißen die Labels auch beispielsweise SUMLO1 oder ERGHI. Der erste Summand wird in den Speicherzellen 250 und 251 übergeben und der zweite Summand in 252 und 253. Da weitere freie Speicherzellen in den ersten 256 Bytes nicht vorhanden sind, wird das Ergebnis in die gleichen Bytes wie der 2. Summand geschrieben.

Im Programm müssen zuerst die niederwertigen und dann die höherwertigen Stellen addiert werden. Nach der ersten Addition enthält das C-Flag den Übertrag, der weitergegeben werden muß. Durch den STA- und LDA-Befehl wird das Flag nicht beeinflußt. Der zweite ADC-Befehl kann es in die Addition einbeziehen. Bei dieser zweiten Addition darf zuvor natürlich mit CLC ein Übertrag nicht gelöscht werden, dies muß nur bei der Addition der LO-Bytes erfolgen. Auch dazu wieder zwei Zahlenbeispiele:

```
POKE 250,16:POKE 251,39:POKE 252,152:POKE 253,58  (10000+15000)
READY.
!GO $1300
AC: ...
READY.
?PEEK(252)+PEEK(253)*256
25000
READY.

POKE 250,0:POKE 251,175:POKE 252,49:POKE 253,212 (44800+54321)
READY.
!GO $1300
AC: ...
READY.
?PEEK(252)+PEEK(253)*256
33585
READY.
```

Aufgabe 6.1 Das Programm zur 16-Bit-Addition soll das Auftreten eines Übertrags anzeigen. Überlegen Sie auch, bei welchem der beiden auftretenden Überträge eine Ausgabe sinnvoll ist.

Als nächstes soll die Subtraktion behandelt werden. Zur Einführung wieder das Beispiel einer binären Subtraktion.

```
 01011101  (93)
-10111010  (186)
1  1   1
----------
 10100011  (163)
```

Zwei einzelne Subtraktionen können wieder zu einer 16-Bit-Subtraktion gekoppelt werden.

```
  00100010     10110001
- 00010010     11011011
   111111 <- 11 1111
 ----------   ----------
  00001111     11010110
```

Die Verarbeitung eines Übertrags durch den Prozessor ist bei der Subtraktion etwas verwirrend. Das C-Flag zeigt nämlich jetzt immer das Gegenteil an. Das Flag ist gesetzt, wenn kein Borgen, wie ein Übertrag bei einer Subtraktion genannt wird, auftrat, und gelöscht beim Entstehen eines solchen.

```
          01011101  (93)              10111010  (186)
        - 10111010  (186)           - 01011101  (93)
C=0 <=  1 1   1           C=1 <= 0 1 111 1
        ----------                  ----------
          10100011  (163)             01011101  (93)
```

Vor einer Subtraktion muß das Übertrags-Flag deshalb auch, im Gegensatz zur Addition, gesetzt werden.

SEC – SEt Carry
– Setze den Übertrag

Zum Ausprobieren werden die Inhalte der Speicherzellen 250 und 251 voneinander subtrahiert. Der Inhalt des C-Flags wird wieder angezeigt. Achtung, hier entspricht diese Anzeige nicht dem Auftreten eines Übertrags bei der Rechnung, sondern genau dem Gegenteil.

```
BSOUT     =     $FFD2
MIN       =     250              ;SPEICHERZELLE DES MINUENDEN
SUB       =     251              ;SPEICHERZELLE DES SUBTRAHENDEN
ERG       =     252              ;SPEICHERZELLE DES ERGEBNISSES
;
          LDA   MIN              ;AKKUMULATOR MIT MINUENDEN LADEN
          SEC                    ;DAS UEBERTRAGS-FLAGS SETZEN
          SBC   SUB              ;DEN SUBTRAHENDEN SUBTRAHIEREN
          STA   ERG              ;DAS ERGEBNIS SPEICHERN
          BCC   ENDE             ;EIN UEBERTRAG ?,NEIN =>
          LDA   #"C"             ;AKKUMULATOR = ASCII-CODE VON "C"
          JSR   BSOUT            ;DAS ZEICHEN AUSGEBEN
ENDE      RTS                    ;DAS PROGRAMM WIEDER VERLASSEN
```

Der Ablauf des Programms wird wieder in Zahlenbeispielen demonstriert:

```
POKE 250,125:POKE 251,50
READY.
!GO $1300
C
AC: ...
READY.
?PEEK (252)
75
READY.

POKE 250,120:POKE 250,200
READY.
!GO $1300
AC: ...
READY.
?PEEK (252)
176
READY.
```

Im letzten Zahlenbeispiel trat bei der Rechnung ein Borgen auf. Da das Übertrags-Flag jedoch immer das Gegenteil anzeigt, erscheint kein Buchstabe "C" auf dem Bildschirm.

Aufgabe 6.2 Schreiben Sie ein Programm, das zwei 16-Bit-Zahlen subtrahiert. Wenn Sie Lust haben, können Sie auch den Inhalt des Übertrags-Flags ausgeben.

Mit dem letzten Programm soll die Verarbeitung von Zahlen in der Zweierkomplement-Darstellung betrachtet werden. Was passiert, wenn zum Beispiel –5 (= 251) minus 10 oder –45 (= 211) minus 120 gerechnet wird?

```
POKE 250,251:POKE 251,10
!GO ...
?PEEK (252)
241
```

```
POKE 250,211:POKE 251,120
!GO ...
?PEEK (252)
91
```

Im ersten Beispiel stimmt das Ergebnis –5 (251) –10 = –15 (241). Aber in Zweierkomplement-Arithmetik ist das zweite Ergebnis falsch: –45 (211) –120 ≠ 91. 91 liegt im Bereich 0 bis127 und muß konsequenterweise als positive Zahl interpretiert werden.

Durch diese Rechnung ist ein in der Zweierkomplement-Arithmetik falsches Ergebnis entstanden. Das mathematisch korrekte Endergebnis (–165) ließe sich außerdem nicht mehr als Zweierkomplementzahl mit 8 Bits darstellen. Wie wird dem Programm mitgeteilt, daß es zu einer derartigen Konstellation gekommen ist, an der Zahl selbst kann es ja nicht erkannt werden?

Im Statusregister gibt es wieder ein Bit, das die Anzeige übernimmt. Es heißt Überlauf-Flag oder kurz V-Flag (oVerflow), da es anzeigt, ob das Ergebnis einer Operation über die Zahlgrenzen der Zweierkomplement-Darstellung hinausgelaufen ist. Das Überlauf-Flag wird auch bei einer Addition gesetzt, falls ein entsprechendes Ereignis auch dort auftritt.

Das Beispielprogramm zur Subtraktion soll jetzt nicht einen Übertrag (C-Flag), sondern einen Überlauf (V-Flag) anzeigen. Für die Entscheidung, ob das Flag gesetzt ist, werden wieder Branch-Befehle benötigt. Diese lauten:

BVC – Branch if oVerflow Clear
– Verzweige, wenn Überlauf gelöscht

BVS – Branch if oVerflow Set
– Verzweige, wenn Überlauf gesetzt

Das Programm sieht dann wie folgt aus:

```
BSOUT   =   $FFD2
MIN     =   250          ;SPEICHERZELLE DES MINUENDEN
SUB     =   251          ;SPEICHERZELLE DES SUBTRAHENDEN
ERG     =   252          ;SPEICHERZELLE DES ERGEBNISSES
;
        LDA MIN          ;AKKUMULATOR MIT MINUENDEN LADEN
        SEC              ;DAS UEBERTRAGS-FLAG SETZEN
        SBC SUB          ;DEN SUBTRAHENDEN SUBTRAHIEREN
        STA ERG          ;DAS ERGEBNIS SPEICHERN
        BVC ENDE         ;EIN UEBERLAUF ?,NEIN =>
        LDA #"V"         ;AKKUMULATOR = ASCII-CODE VON "V"
        JSR BSOUT        ;DAS ZEICHEN AUSGEBEN
ENDE    RTS              ;DAS PROGRAMM WIEDER VERLASSEN
```

Der Vollständigkeit halber sei noch auf einen Befehl hingewiesen, der das Überlauf-Flag gezielt löscht.

CLV – CLear oVerflow
– Lösche das Überlauf-Flag

Ein Befehl zum Setzen des Flags existiert nicht. Der CLV-Befehl wird nur äußerst selten benötigt, da das Überlauf-Flag im Gegensatz zum Übertrags-Flag eine reine Anzeigefunktion hat. Kein Befehl verarbeitet den Inhalt in einer Berechnung oder anderen Operation.

Aufgabe 6.3 Erweitern Sie auch die 16-Bit-Subtraktion um eine Anzeige des Überlaufs. Experimentieren Sie mit verschiedenen Eingaben, und beobachten Sie das Ergebnis.

Aufgabe 6.4 Untersuchen Sie die Bedeutung des Überlauf-Flags bei einem Additionsprogramm. Schreiben Sie dazu ein Programm zur 8-Bit-Addition, und geben Sie den Zustand des Flags auf dem Bildschirm aus.

Noch einmal zurück zur Subtraktion. Die kombinierte Auswertung von Überlauf-, Negativ- und Null-Flag (C-, N-, Z-Flag) erlaubt eine Aussage über die beiden subtrahierten Zahlen.

A	–	B	C	N	Z
100	–	50	1	0	0
40	–	60	0	1	0
35	–	35	1	0	1

An den Flags kann erkannt werden, ob A<B, A=B oder A>B ist. Wie Sie sich bestimmt noch erinnern, führt der Prozessor bei den Vergleichsbefehlen CMP, CPX und CPY intern eine Subtraktion durch, die jedoch nur die Status-Flags dem Ergebnis entsprechend setzt.

In Anbetracht der obigen Zahlenbeispiele kann mit diesen Befehlen nicht nur eine Übereinstimmung zweier Werte festgestellt werden, sondern auch eine größenmäßige Einschätzung ist möglich.

Alle Vergleiche und ihre Ergebnisse sind in folgender Tabelle zusammengestellt. Auch die kombinierten Abfragen kleiner-gleich und größer-gleich sind aufgeführt.

```
LD      #A
CMP     #B
```

	C	N	Z	
A<B	0	1	0	
A=B	1	0	1	
A>B	1	0	0	
A=<B	0	1	0	und genaues Gegenteil
A>=B	1	0	1/0	

Die obige Tabelle weist zwar aus, daß das N-Flag immer den entgegengesetzten Wert wie das C-Flag annimmt, jedoch stimmt dies nur bedingt. In Abfragen sollten Sie deshalb immer das C-Flag prüfen. Dieses Flag wird auch z.B. durch Lade- oder Inkrementierbefehle nicht verändert. Vor einer Abfrage mit dem Branch-Befehl können deshalb noch Register geladen, verändert oder gespeichert werden. In einem Beispiel werden die Inhalte der Speicherzellen 250 und 251 verglichen. Ausgegeben werden die Angaben gleich (=), kleiner (<) und größer null (>). Die Nullbedingung ist einfach mit dem BEQ-Befehl zu erkennen. Die beiden weiteren Bedingungen sind, wie Sie aus der Tabelle ersehen können, dann am C-Flag zu unterscheiden:

```
BSOUT     =     $FFD2
WERTA     =     250          ;VERGLEICHSWERT A
WERTB     =     251          ;VERGLEICHSWERT B
;
     LDA  WERTA              ;VERGLEICHSWERT A LADEN
     CMP  WERTB              ;VERGLEICH MIT WERT B
;
     BNE  GROESSER           ;GLEICH ?,NEIN =>
     LDA  #"="               ;BEI GLEICHHEIT "=" AUSGEBEN
     JMP  BSOUT              ;AUSGABE + PROGRAMM VERLASSEN
;
GROESSER  BCC   KLEINER      ;C-FLAG GESETZT ?,NEIN =>
     LDA  #">"               ;FALLS A>B "<" AUSGEBEN
     JMP  BSOUT              ;AUSGABE + PROGRAMM VERLASSEN
;
KLEINER   LDA   #"<"         ;FALLS A<B ">" AUSGEBEN
     JMP  BSOUT              ;AUSGABE + PROGRAMM VERLASSEN
```

Im Programm werden die einzelnen Bedingungen nacheinander geprüft, und beim Eintreten einer überprüften Bedingung wird das entsprechende Zeichen ausgegeben. Wenn Sie die Tabelle betrachten, werden Sie auch feststellen, daß bei Gleichheit das Übertrags-Flag ebenfalls gesetzt wird.

Damit diese Bedingung nicht als größer eingestuft wird, muß die Abfrage auf Gleichheit durch den BEQ-Befehl vor der Prüfung des C-Flags stattfinden. Aber jetzt zu einigen Zahlenbeispielen:

```
POKE 250,10:POKE 252,47
!GO $1300
<
```

```
POKE 250,201:POKE 251,201
!GO $1300
=

POKE 250,183:POKE 251,25
!GO $1300
>
```

Aufgabe 6.5 Schreiben Sie ein Programm, das den Text "250 kleiner 251" ausgibt, wenn der Inhalt der Speicherzelle 250 kleiner als der Inhalt in Speicherzelle 251 ist.

Spezialfälle von Addition und Subtraktion

Im letzten Kapitel haben Sie die Adressierungsart "nach-indiziert indirekt" kennengelernt. Sie haben auch erfahren, daß beim Zugriff auf größere Tabellen mit dieser Adressierungsart die in der Zero-Page liegende Adresse als Zeiger durch die Tabelle bewegt werden muß. Durch Addition kann der Zeiger verändert werden.

Bei einer 16-Bit-Adresse werden Sie sofort an eine 16-Bit-Addition denken. Meist wird der Zeiger aber innerhalb einer Schleife nur um einige wenige Bytes versetzt. Das HI-Byte ist bei diesen Additionswerten immer null. Es würde unnötig mit addiert, denn warum soll man nul zu einer Zahl addieren?

Um dies zu umgehen, wird folgende Überlegung angestellt: Tritt im niederwertigen Byte kein Übertrag auf, bleibt das höherwertige Byte unverändert. Sollte ein Übertrag auftreten, wird das HI-Byte um eins erhöht, was einem einmaligen Inkrementieren entspricht.

Im Programm kann der Übertrag mit einem Branch-Befehl erkannt werden. Die Erhöhung des HI-Bytes kann der INC-Befehl übernehmen. Die gesamte Routine würde wie folgt aussehen:

```
            LDA     ZEIGER        ;DAS LO-BYTE IN DEN AKKU.
            CLC                   ;DAS C-FLAG LOESCHEN
            ADC     #ADD          ;DAS LO-BYTE ERHOEHEN
            STA     ZEIGER        ;DAS ERGEBNIS SPEICHERN
            BCC     WEITER        ;EIN UEBERTRAG?,NEIN =>
            INC     ZEIGER+1      ;DAS HI-BYTE ERHOEHEN
WEITER      ...
```

Der verwendete Zeiger muß natürlich in den ersten 256 Bytes liegen, damit die nach-indizierte Adressierung angewandt werden kann. Die Routine wird in das Programm zur Textausgabe eingebaut. Der Zeiger wird in 2er-Schritten erhöht, wodurch nur jedes zweite Zeichen des Textes erscheint.

```
BSOUT     =          $FFD2
ZEIGER    =          250         ;ZEIGER IN DEN TEXT
;
          LDA        #<TEXT      ;LO-BYTE DES TEXTANFANGS
          STA        ZEIGER      ;ALS LO-BYTE DES ZEIGERS MERKEN
          LDA        #>TEXT      ;HI-BYTE DES TEXTANFANGS
          STA        ZEIGER+1    ;ALS HI-BYTE DES ZEIGERS MERKEN
          LDY        #0          ;YR=0 (FÜR ADRESSIERUNG)
SCHLEIFE  LDA        (ZEIGER),Y ;ZEICHEN AUS DEM TEXT HOLEN
          BEQ        ENDE        ;ENDMARKE ?,JA =>
          JSR        BSOUT       ;DAS ZEICHEN AUSGEBEN
          LDA        ZEIGER      ;LO-BYTE DES ZEIGERS IN AKKU.
          CLC                    ;C-FLAG LOESCHEN
          ADC        #2          ;DAS LO-BYTE ERHOEHEN
          STA        ZEIGER      ;DAS ERGEBNIS SPEICHERN
          BCC        WEITER      ;EIN UEBERTRAG?,NEIN =>
          INC        ZEIGER+1    ;HI-BYTE DES ZEIGERS ERHOEHEN
WEITER    JMP        SCHLEIFE    ;ZUM ANFANG DER SCHLEIFE SPRINGEN
ENDE                   RTS         ;DAS PROGRAMM VERLASSEN
;
TEXT                   .TEXT "SYBEX-VERLAG",0,0
```

Als Textendmarke werden bei diesem Programm zwei Nullen verwendet. Dies ist notwendig, damit bei Texten mit ungerader Zeichenzahl eine Null am Ende gefunden wird. Bei einem drei Zeichen langen Text z.B. wird das erste und dritte Zeichen ausgegeben.

Eine jetzt folgende Null wird übersprungen. Damit die Ausgabe trotzdem korrekt abgebrochen wird, muß eine weitere Null folgen. Diese wird als 5. Zeichen wieder gelesen. Bei der nach-indizierten indirekten Adressierung wird als Index das Y-Register verwendet. Während des ganzen Programms behält es den Wert Null bei. Es hat eigentlich keine richtige Aufgabe, sondern wird nur benötigt, um die Adressierung zu ermöglichen.

Diesen Umstand findet man oft in Programmen. Mit dem Wert Null als Index wird das 1., 3., 5. Zeichen usw. des Textes ausgegeben. Mit einem Index von eins wird dann folgerichtig das 2., 4., 6. Zeichen usw. auf den Bildschirm geschrieben.

Aufgabe 6.6 Führen Sie die gerade beschriebene Veränderung der Indizes im Programm durch. Variieren Sie zum Experimentieren auch die Schrittweite des Zeigers. Achten Sie dabei aber auf die Endmarke des Textes.

Diese Version der Textausgabe hat den Vorteil, daß der Text beliebig lang sein kann. Bisher wurde als Zeiger in dem Text immer ein Register verwendet. Da die Register nur Werte bis 255 aufnehmen können, durfte der Text auch nur 256 Bytes lang sein. Da jetzt ein 16-Bit-Zeiger verwendet wird, kann der gesamte Speicherbereich adressiert werden. Eine Begrenzung des Textes in seiner Länge ist nicht mehr notwendig.

Um den gesamten Text auszugeben, muß der Zeiger in 1-Byte-Schritten erhöht werden, bzw. die Adresse muß inkrementiert werden. Eine Addition ist für diesen Zweck viel zu aufwendig.

Zwei 8-Bit-Inkrement-Befehle könnten jedoch gekoppelt werden. Wie wird aber ein Übertrag vom LO- zum HI-Byte registriert, die Inkrement-Befehle verändern das Übertrag-Flag ja nicht? Vor dem Entstehen eines Übertrags enthält das LO-Byte den Wert 255.

Durch das nachfolgende Inkrementieren wird das LO-Byte auf null gesetzt, und es würde ein Übertrag entstehen. Dieser parallele Vorgang, Wert Null und möglicher Übertrag, wird ausgenutzt.

Es wird ganz einfach mit dem Befehl BNE dieses Ereignis registriert, und bei seinem Eintreten wird das HI-Byte um eins erhöht. Im Programm sähe dies dann wie folgt aus:

```
            INC         ZEIGER
            BNE         WEITER
            INC         ZEIGER+1
WEITER      ...
```

Dieser Zählvorgang soll in einer Tabelle dargestellt werden. Der Zeiger startet mit dem Wert 4350 (LO: 254, HI: 16)

```
Programmm:      LO   HI    Z-Flag
...            254   16    ?
INC ZEIGER     255   16    0
BNE WEITER     255   16    0
  :
Sprung
  :
INC ZEIGER       0   16    1
BNE WEITER       0   16    1
INC ZEIGER+1     0   17    0

INC ZEIGER       1   17    0
BNE WEITER       1   17    0
  :
Sprung
  :
...
```

Am Ende der Tabelle wurde der Zeiger dreimal erhöht und steht auf dem korrekten Wert 1 + 17 * 256 = 4353 = 4350 + 3. Ins Ausgabeprogramm eingebaut sieht dies wie folgt aus:

```
BSOUT     =          $FFD2
ZEIGER    =          250            ;ZEIGER IN DEN TEXT
;
          LDA        #<TEXT         ;LO-BYTE DES TEXTANFANGS
          STA        ZEIGER         ;ALS LO-BYTE DES ZEIGERS MERKEN
          LDA        #>TEXT         ;HI-BYTE DES TEXTANFANGS
          STA        ZEIGER+1       ;ALS HI-BYTE DES ZEIGERS MERKEN
          LDY        #0             ;YR=0 (FÜR ADRESSIERUNG)
SCHLEIFE  LDA        (ZEIGER),Y     ;ZEICHEN AUS DEM TEXT HOLEN
          BEQ        ENDE           ;ENDMARKE ?,JA =>
          JSR        BSOUT          ;DAS ZEICHEN AUSGEBEN
          INC        ZEIGER         ;DAS LO=BYTE ERHOEHEN
          BNE        WEITER         ;EIN UEBERTRAG?, NEIN =>
          INC        ZEIGER+1       ;HI-BYTE DES ZEIGERS ERHOEHEN
WEITER    JMP        SCHLEIFE       ;ZUM ANFANG DER SCHLEIFE SPRINGEN
ENDE                 RTS            ;DAS PROGRAMM VERLASSEN
TEXT                 .TEXT "SYBEX-VERLAG",0
```

In Programmen werden derartige Zeiger häufig benutzt. Die grobe Schleifenstruktur ist auch in ähnlicher Form immer wieder zu finden. Das Programm kann an einer Stelle noch verbessert werden. Der BNE-Befehl kann gleich zum Schleifenanfang springen, da bei der Marke WEITER auch nichts anderes als ein Rücksprung zum Schleifenanfang erfolgt.

Zwei weitere Operationen mit 16-Bit-Zeigern sollen besprochen werden. Da wäre einmal das Dekrementieren und zweitens der Vergleich, ob eine Endadresse erreicht wurde. Beim Dekrementieren geht man denselben Weg wie beim Inkrementieren. Vor einem möglichen Borgen enthält das LO-Byte den Wert Null.

```
          LDA        ZEIGER         ;DAS Z-FLAG DURCH LADEN SETZEN
          BNE        LOBYTE         ;KOMMT EIN ÜBERTRAG ?,NEIN =>
          DEC        ZEIGER+1       ;DAS HI-BYTE VERMINDERN
LOBYTE    DEC        ZEIGER         ;DAS LO-BYTE VERMINDERN
```

Wie Sie erkennen, hat das Dekrementieren den Nachteil, daß zum Prüfen des LO-Bytes dieses in ein Register geladen werden muß. In einem Programm müssen Sie jeweils ein Register wählen, das nicht benötigt wird. Oft wird aber das LO-Byte in vorherigen oder nachfolgenden Operationen zur Verarbeitung benötigt, womit das Problem auch wieder umgangen wäre.

Jetzt soll der Wert des Zeigers mit einer Endadresse verglichen werden. Dazu muß eine 16-Bit-Subtraktion durchgeführt werden. Diese fällt jedoch relativ einfach aus, da die Ergebnisse nicht gespeichert werden müssen. Bei Gleichheit des Zeigers mit der Endadresse muß die Subtraktion null ergeben. Jeweils nach der Subtraktion des LO-Bytes und des HI-Bytes wird dies mit dem BNE-Befehl geprüft, denn: Warum soll noch das HI-Byte geprüft werden, wenn bereits die Subtraktion des LO-Bytes einen Wert ungleich null ergeben hat?

```
            LDA   ZEIGER       ;AKKU. = LO-BYTE
            SEC                ;UEBERTRAG LOESCHEN
            SBC   #<ENDADR     ;MINUS ENDADRESSE LO-BYTE
            BNE   UNGLEICH     ;NOCH NULL ?,NEIN =>
            LDA   ZEIGER+1     ;AKKU. = HI-BYTE
            SBC   #>ENDADR     ;MINUS ENDADRESSE HI-BYTE
            BNE   UNGLEICH     ;NULL (IDENTISCH) ?,NEIN =>
            ...
UNGLEICH    ...
```

Die Abfrage, ob der Zeiger größer ist als die Endadresse, sieht ähnlich aus:

```
            LDA   ZEIGER       ;AKKU. = LO-BYTE
            SEC                ;UEBERTRAG LOESCHEN
            SBC   #<ENDADR     ;MINUS ENDADRESSE LO-BYTE
            LDA   ZEIGER+1     ;AKKU. = HI-BYTE
            SBC   #>ENDADR     ;MINUS ENDADRESSE HI-BYTE
            BCS   GROESSER     ;NULL (IDENTISCH) ?,NEIN =>
            ...
GROESSER    ...
```

Eigentlich wird mit dieser Routine die Bedingung "größer-gleich" geprüft. Eine korrekte "größer"-Entscheidung ist nur mit etwa dem doppelten Aufwand zu erreichen. Wenn Sie etwas Programmier-Erfahrung gesammelt haben, können Sie sich an diesem Problem versuchen. Für die Abfrage, ob der Zeiger kleiner als die Endadresse ist, muß der BCS-Befehl durch einen BCC-Befehl ersetzt werden.

Aufgabe 6.7 Schreiben Sie ein Programm, das einen beliebig langen Text erst vorwärts und dann wieder rückwärts ausgibt. Ein solches Programm bekommt schon eine stattliche Länge, deshalb ein paar Hilfen:

Für die Vorwärtsausgabe nehmen Sie einfach das letzte Programm zur Textausgabe. Für die Rückwärtsausgabe wird der Zeiger einfach wieder dekrementiert. Die Hilfsmittel kennen Sie jetzt. Die Rückwärtsausgabe benötigt wieder eine Endmarke. Dies kann entweder durch eine Schlußnull (mit .BYTE vor .TEXT einbauen) oder durch Vergleich des Zeigers mit der Anfangsadresse des Textes realisiert werden.

Multiplikation

Zu Beginn gleich eine Aufgabe, in der Sie überlegen können, ob eine Multiplikation sehr effektiv aus Additionen aufgebaut werden kann.

Aufgabe 6.8 Ein Programm soll den Inhalt der Speicherzelle 250 mit drei multiplizieren und das Ergebnis in die Speicherzelle 251 schreiben.

Überlegen Sie, wie die Multiplikation des Inhalts der Zelle 250 mit dem der Zelle 251 realisiert werden könnte. Der Aufbau einer beliebigen Multiplikation aus Additionen würde für die Berechnung viel zu viel Zeit kosten. Man überlege sich nur, daß für die Rechnung 1324 * 735 die Zahl 1234 zu sich selbst 735mal (!!) addiert werden müßte. Für mathematische Berechnungen werden sogar in der Regel größere Zahlen benötigt. Um eine Lösung für das Problem zu finden, betrachten wir die schriftliche Multiplikation zweier Binärzahlen:

```
Multiplikator     Multiplikand
     1011 *       101   (11 * 5)
-------------------
     1011oo
      0000o
       1011
-------------------
   110111       (55)
```

Bei einer Eins im Multiplikanden wird der Multiplikator direkt als Teilergebnis übernommen. Bei einer Null geschieht nichts. Zum Schluß werden alle Teilergebnisse addiert. Daß die eigentliche Multiplikation, im Vergleich zum Dezimalsystem, so einfach ist, liegt an den nur zwei möglichen Ziffern des Binärsystems.

In einem Programm müssen die Stellen der Zahlen (Duplikate des Multiplikators) korrekt addiert werden. In der obigen Rechnung sind hinter jeder Zahl Nullen eingetragen, die die Zahl richtig anordnen. Man erkennt, daß die erste Niederschrift des Multiplikators gegenüber der nächsten Zahl, hier Null, um eine Stelle nach links und gegenüber der letzten einfach um zwei Stellen verschoben ist. Diese vereinfachte Darstellung einer Multiplikation läßt sich in einen Algorithmus fassen: Ist die Ziffer des Multiplikanden eins, addiere den Multiplikator zum Ergebnis. Folgt eine weitere Ziffer im Multiplikanden, schiebe das Ergebnis um eine Stelle nach links und bearbeite diese nächste Ziffer. Das obige Beispiel mit diesem Ablauf sieht wie folgt aus:

```
                1011 * 101
Ergebnis:       0 + 1011 = 1011
Verschieben:    1011 => 10110

                1011 * 101
Ergebnis:       10110 + 0 = 10110
Verschieben:    10110 => 101100

                1011 * 101
Ergebnis:       101100 + 1011 = 110111
```

Noch ein Wort zum Verschieben der Zahl nach links. Dies ist in Wirklichkeit eine Multiplikation mit zwei. Eine Ziffer, die zuvor Bit 0 war und den Stellenwert 1 besaß, ist jetzt in Bit 1 und hat den Stellenwert 2. Gleiches gilt für die weiteren Bits: Bit 1 (2) => Bit 2 (4), Bit 3 => (8), ...

Nimmt man die schriftliche Multiplikation zweier Dezimalzahlen zum Vergleich heran, muß auch hier nach jeder Einzelmultiplikation die Zahl um eine Stelle nach links geschoben werden. Dort entspricht dies einer Multiplikation mit 10.

ASL – Arithmetic Shift Left
– Verschiebe arithmetisch nach links

Dieser Befehl des Prozessors verschiebt den Inhalt der angegebenen Speicherzelle oder den Inhalt des Akkumulators um eine Stelle nach links. Das höchstwertige Bit 7 wird dabei in das Übertrags-Flag geschoben. Das Flag fungiert sozusagen als nicht vorhandenes Bit 8.

In Bit 0 wird eine Null nachgeschoben. Aus dem Namen des Befehls erkennen Sie auch die Bedeutung des Befehls als Multiplikation mit zwei.

```
C-Flag <-  7<---0  <- 0
```

Für ein Programm müssen noch zwei Probleme geklärt werden. Wie wird eine einzelne Ziffer des Multiplikanden ausgelesen und geprüft? Wie wird das Ende der Multiplikation erkannt? Die erste Aufgabe wird auch mit dem ASL-Befehl gelöst. Das höchste Bit, Bit 7, wird beim Verschieben nach links in das Übertrags-Flag transportiert.

Dort kann es mit BCC oder BCS geprüft werden. Bei der nächsten Anwendung von ASL wird wieder Bit 7 in das C-Flag geschoben. Dies ist aber in Wirklichkeit Bit 6 des Multiplikanden, denn beim ersten ASL-Befehl wurde dieses ja nach links in Bit 7 geschoben.

Die gesamte Multiplikation wird abgebrochen, wenn alle Bits des Multiplikanden geprüft wurden. Es wird einfach ein Schleifenzähler eingerichtet.

Im Beispiel wird ein 8-Bit-Multiplikator und -Multiplikand verwendet, folglich zählt der Schleifenzähler von 1 bis 8. Das Ergebnis ist vorerst auch auf 8 Bits beschränkt.

In Abb. 6.1 sehen Sie das Flußdiagramm und nachfolgend das Listing des fertigen Programms:

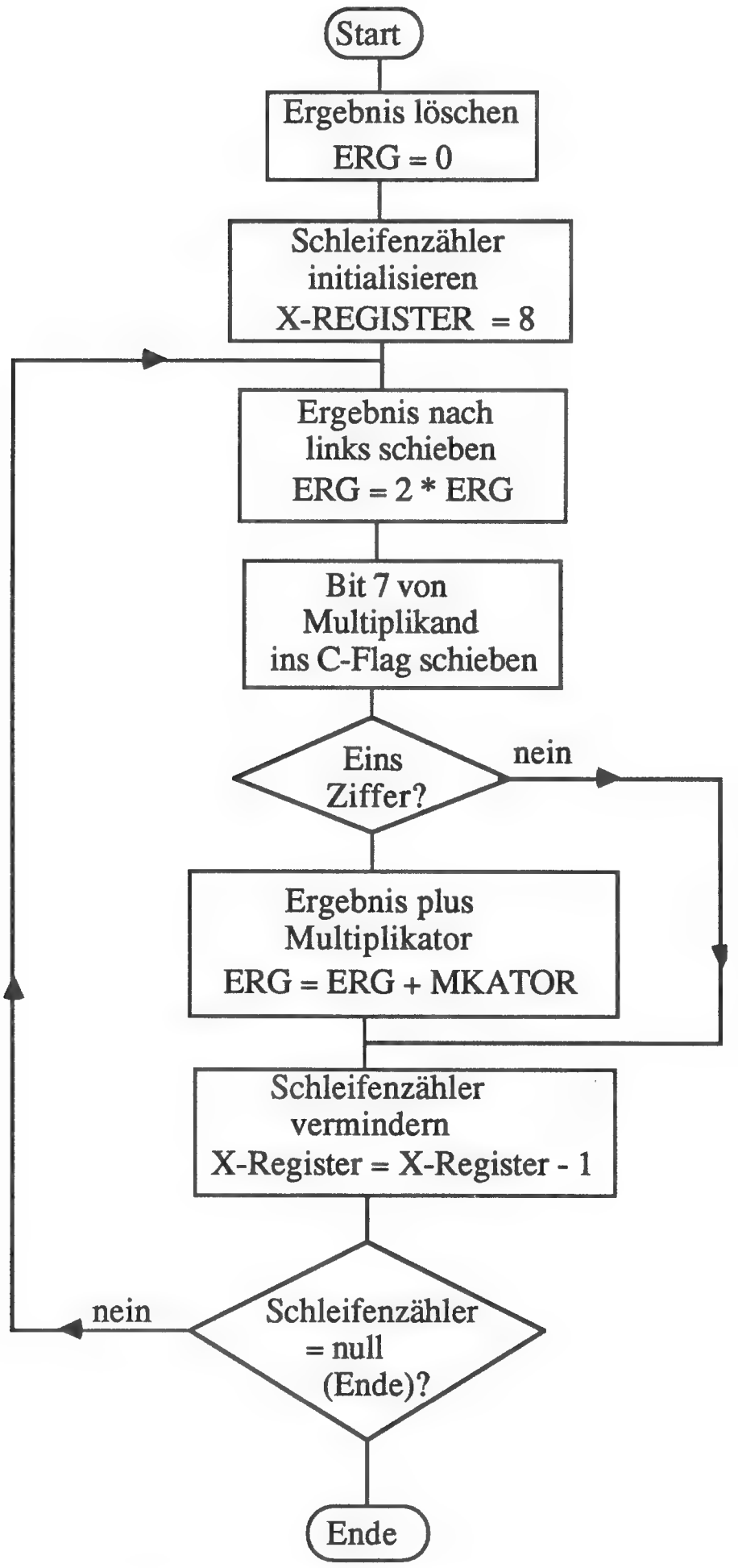


Abb. 6.1: Flußdiagramm des Multiplikationsprogramms

```
MKATOR    =       250         ;SPEICHERZELLE DES MULTIPLIKATORS
MKAND     =       251         ;SPEICHERZELLE DES MULTIPLIKANDEN
ERG       =       252         ;SPEICHERZELLE DES ERGEBNISSES
;
          LDA     #0          ;DAS ERGEBNIS MIT NULL VORBELEGEN
          STA     ERG
          LDX     #8          ;SCHLEIFENZAEHLER MIT 8 STARTEN
SCHLEIFE  ASL     ERG         ;DAS ERGEBNIS NACH LINKS SCHIEBEN
          ASL     MKAND       ;BIT 7 VON MKAND INS C-FLAG SCHIEBEN
          BCC     WEITER      ;WAR ES EINE EINS?, NEIN =>
          LDA     ERG         ;ERGEBNIS IN DEN AKKU.
          CLC                 ;DAS UEBERTRAGSFLAG LOESCHEN
          ADC     MKATOR      ;DEN MULTIPLIKATOR ADDIEREN
          STA     ERG         ;UND DAS ERGEBNIS SPEICHERN
WEITER    DEX                 ;DEN SCHLEIFENZAEHLER VERMINDERN
          BNE     SCHLEIFE    ;NULL (ENDE) ERREICHT?,NEIN =>
          RTS                 ;DAS PROGRAMM VERLASSEN
```

Das Ergebnis wird zu Beginn des Schleifenrumpfs verschoben. Damit ist gewährleistet, daß nachfolgend noch eine Stelle des Multiplikanden bearbeitet wird. Beim ersten Aufruf des Schleifenrumpfs ist dieser Vorgang zwar überflüssig, aber da das Ergebnis mit dem Wert Null startet, ändert es am Endprodukt der Multiplikation nichts. Zum Ausprobieren des Programms ein paar Zahlenbeispiele:

```
POKE 250,10:POKE 251,12
!GO $1300
?PEEK (252)
120

POKE 250,17:POKE 251,9
!GO $1300
?PEEK (252)
153

POKE 250,25:POKE 251,53
!GO $1300
?PEEK (252)
45
```

Aus den letzten Zahlen erkennen Sie, daß ein 16-Bit-Ergebnis dringend erforderlich ist. Mit zwei 8-Bit-Faktoren sind Ergebnisse bis 255*255=65025 möglich. Bei der Addition des Multiplikators kann auf eine 16-Bit-Addition zurückgegriffen werden.

Für das Links-Verschieben müssen zwei einzelne Schiebeoperationen gekoppelt werden. Bit 7 des LO-Bytes, das ins Übertrags-Flag geschoben wird, muß

in einer zweiten Operation ins Bit 0 des HI-Bytes geschoben werden. Der ASL-Befehl kann diese Aufgabe nicht erfüllen, denn dieser schiebt ja grundsätzlich eine Null in Bit 0 nach.

ROL – ROtate Left
– rotiere einen Wert nach links

Dieser Befehl schiebt im Gegensatz zum ASL-Befehl das Übertragsflag in Bit 0 nach. Bei einem weiteren ROL-Befehl würde das Carry-Flag, d.h. das ehemalige Bit 7, in Bit 0 geschoben. Nach neun Rotations-Befehlen erhielte man wieder den ursprünglichen Wert. In einer Grafik sieht die Rotation wie folgt aus:

C-Flag <- 7<---0 <- C-Flag

Mit dieser Erklärung ließe sich der ASL-Befehl als eine Folge aus einem CLC- und einem ROL-Befehl interpretieren. Eine 16-Bit-Schiebeoperation sieht jetzt wie folgt aus:

```
ASL  ERGLO       ;DAS LO-BYTE VERSCHIEBEN (BIT7->C)
ROL  ERGHI       ;DAS HI-BYTE VERSCHIEBEN (C->BIT0)
```

Die Befehle lassen sich direkt ins Programm übernehmen, das dann wie folgt aussieht:

```
MKATOR    =      250         ;SPEICHERZELLE DES MULTIPLIKATORS
MKAND     =      251         ;SPEICHERZELLE DES MULTIPLIKANDEN
ERGLO     =      252         ;LO-BYTE DES ERGEBNISSES
ERGHI     =      253
;HI-BYTE DES ERGEBNISSES
;
          LDA    #0          ;DAS ERGEBNIS MIT NULL VORBELEGEN
          STA    ERGLO
          STA    ERGHI
          LDX    #8          ;SCHLEIFENZAEHLER MIT 8 STARTEN
SCHLEIFE  ASL    ERGLO       ;DAS ERGEBNIS NACH LINKS SCHIEBEN
          ROL    ERGHI
          ASL    MKAND       ;BIT 7 VON MKAND INS C-FLAG SCHIEBEN
          BCC    WEITER      ;WAR ES EINE EINS?, NEIN =>
          LDA    ERGLO       ;LO-BYTE-ERGEBNIS IN DEN AKKU.
          CLC                ;DAS UEBERTRAGSFLAG LOESCHEN
          ADC    MKATOR      ;DEN MULTIPLIKATOR ADDIEREN
          STA    ERGLO       ;DAS LO-BYTE SPEICHERN
          LDA    ERGHI       ;HI-BYTE DES ERGEBNISSES LADEN
          ADC    #0          ;EINEN EVTL. UEBERTRAG ADDIEREN
          STA    ERGHI       ;DAS HI-BYTE SPEICHERN
WEITER    DEX                ;DEN SCHLEIFENZAEHLER VERMINDERN
          BNE    SCHLEIFE    ;NULL (ENDE) ERREICHT?,NEIN =>
     RTS                     ;DAS PROGRAMM VERLASSEN
```

Aufgabe 6.9 Im letzten Abschnitt haben Sie einige Spezialfälle der 16-Bit-Addition kennengelernt. Dort wurden sie auf Zero-Page-Zeiger angewandt. Versuchen Sie das Multiplikationsprogramm mit diesen Routinen zu verbessern.

Noch einmal zurück zum ASL-Befehl. Wird in Programmen eine Multiplikation mit 2, 4, 8 oder anderen Zweierpotenzen benötigt, wird auf diesen Befehl zurückgegriffen. Eine wiederholte Anwendung liefert die entsprechenden Ergebnisse. Hier z.B. die Multiplikation des Inhalts der Speicherzelle 250 mit 8. Zur Beschleunigung wird das eigentliche Verschieben im Akkumulator abgewickelt:

```
LDA 250         ;AKKUMULATOR LADEN
ASL A           ;*2
ASL A           ;*2 (INSGESAMT *4)
ASL A           ;*2 (INSGESAMT *8)
STA 250         ;DAS ERGEBNIS SPEICHERN
RTS             ;DAS PROGRAMM VERLASSEN
```

Sie sehen hier auch, wie der Akkumulator von den Schiebebefehlen adressiert wird. Bei dem Buchstaben "A" im Operanden handelt es sich nicht um eine Adresse, sondern nur um einen Hinweis für den Assembler. Im Objektcode ist der Befehl nur ein Byte lang. Die Angabe "A" kann bei EDASS auch entfallen.

Außer diesen Zweierpotenzen lassen sich auch andere Faktoren leicht aufbauen, als Beispiel die Multiplikanden 5 und 7. Zuerst die mathematische Grundlage und dann das Programm:

$$5 * X = 4 * X + X$$

```
LDA  250              ;DEN AKKUMULATOR LADEN
ASL  A                ;*2
ASL  A                ;*2 (INSGESAMT *4)
CLC                   ;UEBERTRAGS-FLAG LOESCHEN
ADC  250              ;DEN WERT ADDIEREN (INS. *5)
STA  250              ;DAS ERGEBNIS SPEICHERN
RTS                   ;DAS PROGRAMM VERLASSEN
```

$$7 * X = 6 * X + X = 3 * X * 2 + X = (2 * X + X) * 2 + X$$

```
LDA  250              ;DEN AKKUMULATOR LADEN
ASL  A                ;*2
CLC                   ;DAS UEBERTRAGS-FLAG LÖSCHEN
ADC  250              ;DEN WERT ADDIEREN (INSGESAMT.*3)
ASL  A                ;*2 (INSGESAMT *6)
CLC                   ;DAS UEBERTRAGS-FLAG LOESCHEN
ADC  250              ;DEN WERT ADDIEREN (INSGESAMT *7)
STA  250              ;DAS ERGEBNIS SPEICHERN
RTS                   ;DAS PROGRAMM VERLASSEN
```

Beide Programme werden aus der Kombination von Multiplikationen mit zwei und Additionen aufgebaut. Faktoren bis etwa 10 sollten mit dieser Methode aufgebaut werden. Für größere Werte werden nur Nachteile eingehandelt, und man sollte nach anderen Lösungen suchen.

Aufgabe 6.10 Schreiben Sie zwei Programme, die Multiplikationen mit 3 und 10 nach der oben beschriebenen Methode durchführen.

Division

Zur Herleitung der Division wird wieder der gleiche Weg beschritten wie bei der Multiplikation, also hier wieder das schriftliche Dividieren zweier Binärzahlen:

```
Dividend    Divisor
  100011 :  111   = 0101          (35 : 7 = 5 )
  111 ---------------- ↑
  - 111 ----------------↑
  ------
    11 ----------------- ↑
  - 111 ----------------↑
  ------
     0
```

In die Rechnung sind in kursiver Schrift die Subtraktionsversuche eingetragen, die eine Ziffer Null im Endergebnis erzeugen. Ein klares Ablaufschema ist aus der Berechnung noch nicht abzuleiten, darum die ganze Rechnung nochmals in einer ausführlicheren Form:

```
 100011    100011    000111    000111
-111000  -  11100  -  1110   -  111
--------  --------  --------  --------
 neg.      000111    neg.      000111
  ↓         ↓         ↓         ↓
  0         1         0         1
```

Aus der Darstellung sind zwei Sachverhalte zu erkennen. Der Divisor wird zu Beginn der Division mehrere Stellen nach links geschoben. Von Subtraktion zu Subtraktion wandert er wieder nach rechts, bis er den ursprünglichen Wert erreicht hat. Danach ist die Division beendet. Im Endergebnis entsteht eine Eins, wenn die Subtraktion ein positives Ergebnis liefert, und eine Null, wenn die Subtraktion ein negatives Ergebnis ergibt und in diesem Fall nicht durchgeführt wird.

Für ein Programm muß geklärt werden, wie weit zu Beginn der Divisor nach links geschoben wird. Es könnte so lange geschoben werden, bis dieser größer ist als der Dividend. Da im Assemblerprogramm die Stellenzahl fest begrenzt ist, in diesem Fall auf 8 Bit, kann der Divisor auch ganz nach links bis zum Erreichen von Bit 7 geschoben werden.

Zwar ist damit vielleicht der erste Subtraktionswert sehr viel größer als der Dividend, dies aber beeinflußt das Endergebnis nicht. Ein Zähler muß die Anzahl der Schiebevorgänge notieren, damit bekannt ist, wie oft später nach rechts geschoben werden darf und wie oft eine Subtraktion versucht werden darf. Für das Programm fehlt noch ein Befehl zum Rechts-Verschieben.

LSR – Logical Shift Right
– Verschiebe logisch nach rechts

Der Befehl schiebt den Inhalt einer Speicherzelle oder des Akkumulators nach rechts. Bit 0 wird in das Übertrags-Flag geschoben und Bit 7 wird mit einer Null aufgefüllt.

0 -> 7--->0 -> C-Flag

ROR – ROtate Right
– rotiere einen Wert nach rechts

Dies ist der entsprechende Rotationsbefehl. Auch hier wird in Bit 7 der Inhalt des C-Flags nachgeschoben und Bit 0 ins C-Flag geschoben. Wie beim ROL-Befehl erhält man nach neun Rotationen wieder den Ausgangswert.

C-Flag -> 7--->0 -> C-Flag

Das Flußdiagramm ist in Abb. 6.2 zu sehen, nachfolgend der Programmtext:

```
DIVIDEND =      250          ;SPEICHERZELLE DES DIVIDENDEN
DIVISOR  =      251          ;SPEICHERZELLE DES DIVISORS
ERG      =      252          ;SPEICHERZELLE DES ERGEBNISSES
;
         LDA    #0           ;DAS ERGEBNIS MIT NULL BELEGEN
         STA    ERG
         LDX    #0           ;DEN ZAEHLER MIT DEM WERT 0 STARTEN
SCHIEBEN INX                 ;DEN ZAEHLER UM EINS ERHOEHEN
         LDA    DIVISOR      ;DEN DIVISOR IN DEN AKKU. LADEN
         BMI    SCHLEIFE     ;DIVISOR AN ZAHLGRENZE ?, JA =>
         ASL    DIVISOR      ;DEN DIVISOR NACH LINKS SCHIEBEN
         JMP    SCHIEBEN     ;ZUM ANFANG DER SCHIEBESCHLEIFE
;
```

```
SCHLEIFE ASL   ERGEBNIS     ;DAS ERGEBNIS NACH LINKS SCHIEBEN
         LDA   DIVIDEND     ;DEN DIVIDENDEN IN AKKUMULATOR HOLEN
         CMP   DIVISOR      ;DIVIDEND MIT DIVISOR VERGLEICHEN
         BCC   WEITER       ;DIVIDEND < DIVISOR?, JA =>
         SEC                ;DAS UEBERTRAGS-FLAG SETZEN
         SBC   DIVISOR      ;ENDGUELTIGE SUBTRAKTION DURCHFUEHREN
         STA   DIVIDEND     ;DAS SUBTRAKTION-ERGEBNIS SPEICHERN
         INC   ERG          ;ENDERGEBNIS + 1
WEITER   LSR   DIVISOR      ;DIVISOR NACH RECHTS SCHIEBEN
         DEX                ;DEN ZAEHLER UM EINS VERMINDERN
    BNE  SCHLEIFE           ;NULL (ENDE)?, NEIN =>
    RTS                     ;DAS PROGRAMM VERLASSEN
```

In der ersten Schleife wird der Divisor nach links geschoben, bis bei ihm Bit 7 gesetzt ist. Aus dem ersten Abschnitt dieses Kapitels wissen Sie, daß negative Zahlen durch ein gesetztes Bit 7 gekennzeichnet sind. Für die Abfrage, ob das Ende des Schiebevorgangs erreicht ist, wird diese Eigenschaft mit dem BMI-Befehl natürlich ausgenutzt. Parallel zählt das X-Register die Anzahl der Schiebevorgänge.

Die eigentliche Division dürfte klar sein. Interessant ist nur, wie die einzelnen Ziffern im Ergebnis erzeugt werden. Vor jeder Subtraktion werden diese nach links geschoben. Bit 0 nimmt immer den Platz der zu berechnenden Ziffer ein. Durch den ASL-Befehl wird das Bit mit null gefüllt. Entfällt die Subtraktion, befindet sich bereits die richtige Ziffer an dieser Stelle. Wird sie jedoch durchgeführt, erhöht der INC-Befehl das gesamte Ergebnis um eins.

Da Bit 0 null enthält, kommt dies einem Setzen des Bits gleich (11010 +1 = 11011, 101010 + 1=101011).

Probieren Sie das Programm einmal aus. Dividend und Divisor werden mit POKE in die Speicherzellen 250 und 251 geschrieben, und das Ergebnis wird mit PEEK aus 252 gelesen. Zum Experimentieren geben Sie auch als Divisor null ein. Mathematisch ist dieser Quotient nicht definiert, und auch jeder Taschenrechner würde eine Fehlermeldung liefern. Was macht das Programm?

Na, haben Sie es ausprobiert? Der Computer ist, wie man so schön sagt, "abgestürzt". Er nimmt keine Eingaben mehr an. Drücken Sie RUN/STOP und RESTORE, um wieder weiterarbeiten zu können. Das Problem liegt in der ersten Schleife. Der Divisor wird so lange nach rechts geschoben, bis das Bit 7 gesetzt ist. Natürlich muß dieses durch die Schiebevorgänge aus einem der Bits 0 bis 6 stammen. Bei einem Divisor von null ist aber überhaupt kein Bit gesetzt. Es kann also auch nie eines in die höchste Stelle gelangen. Die Endbedingung der Schleife kann nie erfüllt werden, und folglich wird die Schleife auch nie beendet. Der Computer sitzt in einer Endlosschleife fest.

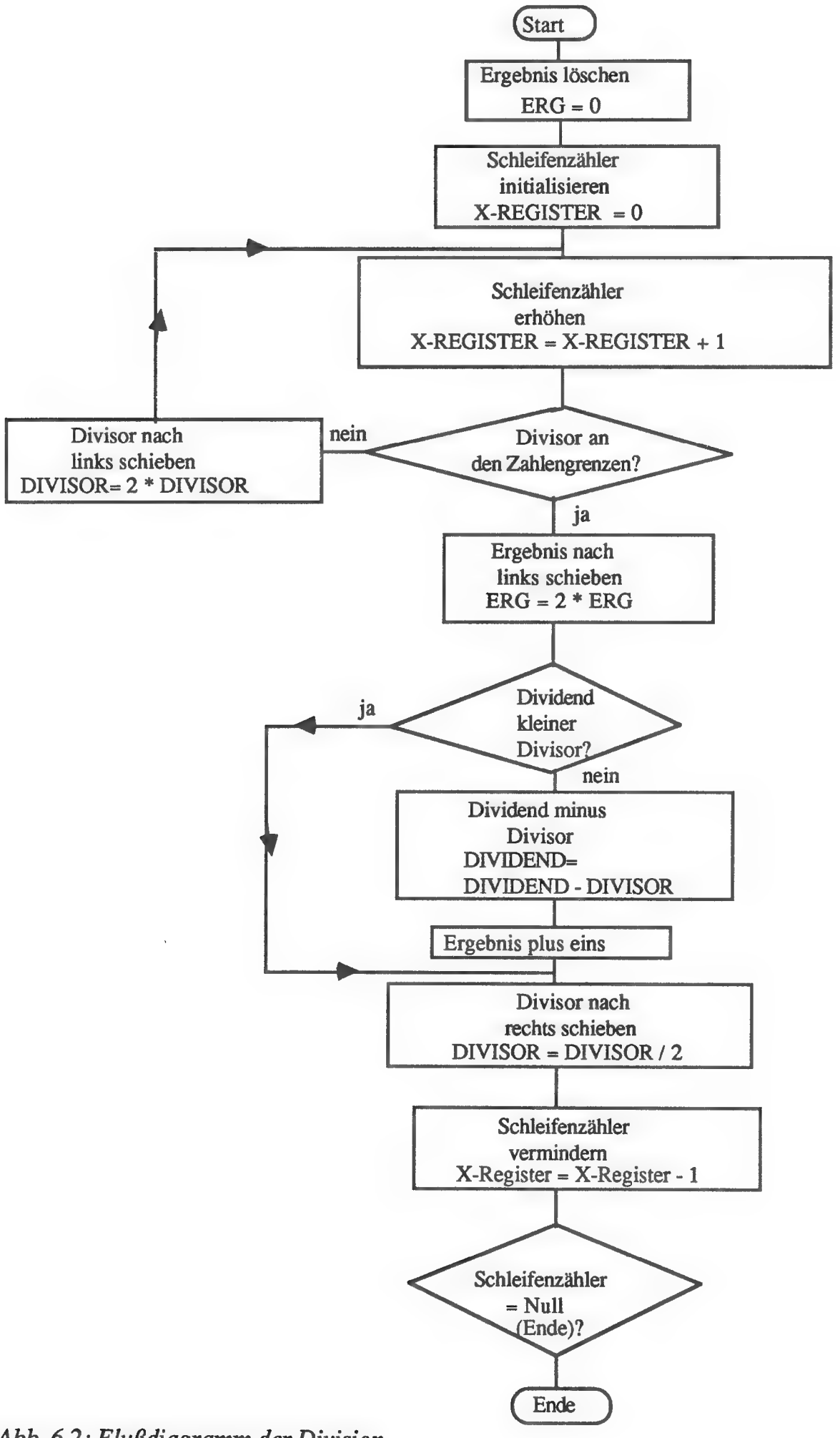


Abb. 6.2: Flußdiagramm der Division

Aufgabe 6.11 Fügen Sie in das Divisionsprogramm eine Abfrage ein, die prüft, ob der Divisor den Wert Null hat. Es soll dann z.B. ein "E" als Fehlermeldung ausgegeben werden.

Leider können, wie bei der Multiplikation, Divisionen mit einem kleinen Faktor nicht auf Schiebe- und Subtraktionsbefehl zurückgeführt werden. Lediglich Divisionen durch 2, 4, 8 und andere Zweierpotenzen können durch Hintereinanderschaltungen von Rechtsverschiebungen realisiert werden.

16-Bit-Zahlen werden durch Koppeln des LSR- mit dem ROR-Befehl nach rechts geschoben. Da die Bits hier von der höherwertigen zur niederwertigen Stelle geschoben werden, muß natürlich auch zuerst das HI-Byte mit LSR und dann das LO-Byte mit ROR verschoben werden.

Dezimal-Arithmetik

Sie werden sich wahrscheinlich fragen, was so besonders an "Dezimal-Arithmetik" ist. Die Programme in den letzten Abschnitten haben doch einwandfreie Ergebnisse geliefert. Dies ist aber nur dem Umstand zu verdanken, daß POKE eine Dezimalzahl in die binäre Darstellung umwandelt und PEEK das binäre Ergebnis in eine Dezimalzahl.

Eine Ausgabe des Ergebnisses direkt durch das Assemblerprogramm ist nur mit einem längeren Programm möglich.

Für einfache Berechnung hilft der sogenannten Dezimal-Modus des Prozessors weiter. Ein Zahlenwert wird dann von den Befehlen ADC und SBC nicht wie ein binärer Wert, sondern als zwei Dezimalziffern interpretiert.

Die 8 Bit eines Bytes werden dazu in Gruppen zu je vier Bits, also einem Nibble, aufgeteilt. In den vier niederwertigen Bits steht auch die niedrigere Dezimalziffer und entsprechend in den anderen vier Bits die höherwertige.

```
0110 1001      0001 0000      1001 1001
  ↓    ↓         ↓    ↓         ↓    ↓
  6    9         1    0         9    9
```

Neun ist also der Maximalwert, den ein Nibble enthalten darf. Ein Byte kann die Zahlen von 0 bis 99 darstellen. Solche Zahlen werden als BCD-Zahlen bezeichnet (Binär-Codierte-Dezimal-Zahlen).

SED – SEt Decimal mode
– schalte den Dezimal-Modus ein

CLD – CLear Decimal mode
– schalte den Dezimal-Modus ab

Diese beiden Befehle schalten den Prozessor zwischen beiden Modi hin und her. Angezeigt wird die Betriebsart im Statusregister im Dezimal-Flag oder kurz D-Flag. Die Additions- und Subtraktionsbefehle arbeiten mit BCD-Zahlen. Es kann die BCD-Darstellung z.B. der Zahlen 17 und 33 direkt addiert werden:

```
  0001 0111 (17)
+ 0011 0011 (33)
-----------------
  0101 0000 (50)
```

Zum Vergleich hier die Addition dieser zwei Bytes, interpretiert als Binärzahlen:

```
  00010111  (23)
+ 00110011  (51)
--------------
  01001010  (74)
```

Schon allein durch die Umwandlung der Binärzahlen in Dezimal-Zahlen ergeben sich andere Werte. Aber auch die Binärziffern des Endergebnisses unterscheiden sich vom Dezimalmodus. Wird in einem Programm mit diesem Modus gearbeitet, muß der Programmierer das An- und Abschalten genau einhalten. Noch ein wichtiger Punkt: Alle Befehle außer ADC und SBC werden durch den Dezimal-Modus nicht beeinflußt! Inkrement-, Dekrement- oder Vergleichsbefehle arbeiten weiterhin mit der binären Interpretation. Eine BCD-Zahl kann nicht durch einen Inkrementierbefehl um eins erhöht werden. Jetzt ein kleines Programmbeispiel, das die Inhalte der Speicherzellen 250 und 251 als BCD-Zahlen interpretiert.

```
SUM1    =       250         ;1. SUMMAND
SUM2    =       251         ;2. SUMMAND
ERG     =       252         ;ERGEBNIS
;
        SED                 ;DEZIMAL-MODUS EINSCHALTEN
        LDA     SUM1        ;DEN 1. SUMMANDEN IN AKKUMULATOR LADEN
        CLC                 ;DAS UEBERTRAGS-FLAG LOESCHEN
        ADC     SUM2        ;DEN 2. SUMMANDEN ADDIEREN
        STA     ERG         ;DAS ERGEBNIS SPEICHERN
        CLD                 ;DEN DEZIMAL-MODUS AUSSCHALTEN
        RTS                 ;DAS PROGRAMM VERLASSEN
```

Es ist wichtig, daß der Dezimal-Modus am Programmende wieder abgeschaltet wird. Alle Routinen im ROM des C128 würden falsche Ergebnisse liefern, da sie erwarten, daß der Dezimal-Modus nicht aktiv ist.

Das Programm wird wieder mit einigen Zahlenbeispielen getestet. Bei der Zahleneingabe wird ein kleiner Trick angewandt. Die höherwertigen vier Bits haben den Stellenwert 16. Die höhere Dezimalziffer wird deshalb auch durch den Ziffernwert mal 16 errechnet.

```
POKE 250,1*16+7:POKE 251,3*16+3
!GO $1300
?PEEK (252)
80
```

Die Zahl 80 entspricht der BCD-Zahl 50.

Die Ausgabe des Ergebnisses ist aber auch leicht durch das Assemblerprogramm selbst vorzunehmen. Die Ziffern 0 bis 9 haben den ASCII-Code 48 bis 58. Durch Ziffernwert + 48 (z.B. 0 + 48 = 48, 5 + 48= 53) wird direkt in den ASCII-Code umgerechnet.

Wie wird aber eine einzelne Ziffer isoliert? Bei der oberen Stelle ist dies ganz einfach: Es wird die gesamte Zahl viermal nach rechts verschoben. Die unterste Stelle ist aus dem Byte herausgeschoben, und die obere Stelle liegt in den unteren vier Bits. Durch Addition von 48 erhält man den ASCII-Code der Ziffer. Für die untere Ziffer aber müßten die oberen vier Bits sozusagen "entfernt" oder "ausgelöscht" werden. Das Zauberwort hierfür heißt maskieren bzw. ausblenden.

AND – logical AND
– logisch UND-verknüpfen

Vielleicht sind Sie schon einmal früher auf diese elementare Operation gestoßen. Die elektronische Nachbildung hilft mit, Computerbausteine wie den Prozessor aufzubauen. Durch die logische Funktion werden zwei Bits miteinander verknüpft. Die folgende Wertetabelle zeigt die möglichen Ergebnisse:

A	B	A UND B
0	0	0
1	0	0
0	1	0
1	1	1

Als kleine Regel kann man sich merken, daß im Endergebnis nur eine Eins zustandekommt, wenn Bit A UND Bit B den Wert Eins enthalten. Bei zwei 8-Bit-Zahlen werden jeweils die gleichwertigen Bits miteinander verknüpft:

```
  11011011      10100101      10111101      10101111
UND 01010101  UND 11001011  UND 11111111  UND 00000000
----------    ----------    ----------    ----------
  01010001      10000001      10111101      00000000
```

In den letzten zwei Beispielen sehen Sie Spezialfälle. Lauter Einsen verändern den zweiten Zahlenwert nicht. Eins trifft auf Eins und ergibt eine Eins; Null trifft auf Eins, was eine Null zur Folge hat. Lauter Nullen dagegen löschen eine Zahl. Null und Eins treffen auf Null, und es ergibt immer eine Null. Genau diese zwei Effekte werden zum Maskieren ausgenutzt. In der Maske wird bei den zu löschenden Bits eine Null eingetragen und bei den anderen, die unverändert bleiben sollen, steht eine Eins.

```
  10011101      10100110      10101010
UND 00001111  UND 11110000  UND 00111100
----------    ----------    ----------
  00001101      10100000      00101000
```

Wird in der Maske nur ein einzelnes Bit gesetzt, kann auf ein einzelne Stelle zugegriffen werden. Für die Ausgabe der BCD-Zahl wird eine Maske verwendet, die die obere Stelle entfernt. Das Additionsprogramm mit Ausgabe sieht dann wie folgt aus:

```
BSOUT =  $FFD2
SUM1     =          250       ;1. SUMMAND
SUM2     =          251       ;2. SUMMAND
ERG      =          252       ;ERGEBNIS
;
         SED                  ;DEZIMAL-MODUS EINSCHALTEN
         LDA        SUM1      ;DEN 1.SUMMANDEN IN AKKUMULATOR LADEN
         CLC                  ;DAS UEBERTRAGS-FLAG LOESCHEN
         ADC        SUM2      ;DEN 2.SUMMANDEN ADDIEREN
         STA        ERG       ;DAS ERGEBNIS SPEICHERN
         CLD                  ;DEN DEZIMAL-MODUS AUSSCHALTEN ;
         LSR        A         ;DIE OBERE STELLE VERSCHIEBEN
         LSR        A
         LSR        A
         LSR        A
         CLC                  ;DEN UEBERTRAG LOESCHEN
         ADC        #48       ;DEN ASCII-CODE ERRECHNEN
         JSR        BSOUT     ;UND DIE ZIFFER AUSGEBEN
         LDA        ERG       ;DAS ERGEBNIS IN DEN AKKU. LADEN
         AND        #15       ;DIE UNTERE STELLE ISOLIEREN
         CLC                  ;DEN UEBERTRAG LOESCHEN
         ADC        #48       ;DEN ASCII-CODE ERRECHNEN
         JMP        BSOUT     ;UND ZIFFER AUSGEBEN+PROGRAMM VERLASSEN
```

Im Programm ist es wichtig, daß der Dezimal-Modus vor der Ausgabe abgeschaltet wird. Die Umrechnung in den ASCII-Code, die den ADC-Befehl benötigt, würde sonst falsche Ergebnisse liefern.

```
POKE 250,1 * 16 + 7:POKE 251,3 * 16 + 3
!GO $1300
50

POKE    250,3:POKE 251,4
!GO $1300
07
```

Aufgabe 6.12 Wie im letzten Zahlenbeispiel zu sehen ist, gibt das Programm eine Führungsnull aus. Erweitern Sie das Programm um eine Abfrage, die die Ausgabe einer Null an der oberen Stelle unterbindet.

Aufgabe 6.13 Schreiben Sie ein Programm für die Subtraktion im Dezimal-Modus. Auch hier soll die Ausgabe des Ergebnisses erfolgen.

Kapitel 7

Bitmanipulationen

UND, ODER und EXKLUSIV-ODER

Unter dem Namen "Bit-Manipulationen" werden alle Operationen zusammengefaßt, die es erlauben, ein einzelnes Bit zu bearbeiten.

Einen wichtigen Befehl für diese Zwecke haben Sie bereits im letzten Kapitel kennengelernt. Mit AND werden einzelne Bits gezielt gelöscht.

Die übrigen Bits werden auf diese Weise z.B. isoliert und aus dem Byte herausgetrennt. Noch einmal ein paar Beispiele:

```
  10101110        10110101        01011100
UND 10011001    UND 00001111    UND 10101110
----------      ----------      ----------
  10001000        00000101        00001100
```

ORA – logical OR with Accumulator and memory
– ODER-Verknüpfung von Akkumulator und Speicher

In der Anwendung (nicht in der logischen Funktion) hat dieser Befehl eine gegenteilige Wirkung wie der AND-Befehl. Zwei einzelne Bits werden wie folgt zu einem Ergebnis verknüpft:

A	B	A UND B
0	0	0
1	0	1
0	1	1
1	1	1

Es entsteht immer dann eine Eins, wenn Bit A ODER Bit B den Wert Eins besitzt. Auch hier werden wieder bei 8-Bit-Zahlen zwei gleichwertige Stellen miteinander verknüpft:

```
  10101100      01010011      10111001      10010101
ODER01001001  ODER 11000101  ODER 11111111 ODER 00000000
--------      --------      --------      --------
  11101101      11010111      11111111      10010101
```

Die zwei letzten Beispiele zeigen wieder Spezialfälle. Lauter Einsen ergeben nur Einsen, und lauter Nullen verändern die Zahl nicht. Im Assemblerprogramm wird der Befehl vor allem verwendet, um Bits gezielt zu setzen. In einem Beispiel wird dies ausgenutzt, um zwei Ziffern, deren Werte in den Speicherzellen 250 und 251 abgelegt sind, zu einer Dezimalzahl in BCD-Darstellung zusammenzufügen. Die höherwertige Ziffer muß in die oberen vier Bits des Ergebnisses transportiert werden und die niederwertige in die unteren vier Bits.

```
ZIFFERHI  =       250       ;HOEHERWERTIGE ZIFFER
ZIFFERLO  =       251       ;NIEDERWERTIGE ZIFFER
ERG       =       252       ;ERGEBNIS
;
          LDA     ZIFFERHI  ;DIE OBERE ZIFFER IN DEN AKKUMULATOR
          ASL     A         ;VIERMAL NACH LINKS SCHIEBEN
          ASL     A
          ASL     A
          ASL     A
          ORA     ZIFFERLO  ;DIE UNTERE ZIFFER EINBLENDEN
          STA     ERG       ;DAS ERGEBNIS SPEICHERN
          RTS               ;DAS PROGRAMM VERLASSEN
```

Sollen die Ziffern 5 und 7 zu einer BCD-Zahl verbunden werden, führt das Programm im Akkumulator folgende Schritte aus:

00000101	5 laden
00001010	nach links schieben (1)
00010100	nach links schieben (2)
00101000	nach links schieben (3)
01010000	nach links schieben (4)
01010111	7 durch ODER-Verknüpfung einblenden

Durch die Kombination von AND und ORA läßt sich ein Byte beliebig "zusammenschnippeln".

```
        11110010
UND     11000011
--------
        11000010
ODER    00101000
--------
        11101010
```

Hier wurden z.B. die Bits 2-5 mit AND "herausgeschnitten, und statt dessen wurde die Ziffernfolge 1010 mit ORA "hineingeklebt".

EOR – Exclusive-OR with accumulator and memory
– EXKLUSIV-ODER-Verknüpfung von Akkumulator und Speicher

Dies ist die dritte logische Verknüpfung, die der 6502- bzw. 8502-Prozessor anbietet. Die Wertetabelle sieht wie folgt aus:

A	B	A EXOR B
0	0	0
1	0	1
0	1	1
1	1	1

Es entsteht nur eine Eins als Ergebnis, wenn Bit A und Bit B verschiedene Werte enthalten. Dies ist eine einfache Beschreibung der logischen Operation. In Texten wird der Name Exklusiv-Oder mit EXOR oder XOR abgekürzt.

```
  10101011        11010110        11001101        10110101
EXOR11010010  EXOR 01010101  EXOR11111111  EXOR 00000000
--------        --------        --------        --------
  01111001        10000011        00110010        10110101
```

Wie bei den anderen Verknüpfungen ergeben lauter Nullen bzw. Einsen spezielle Effekte. Nullen lassen einen Wert unberührt, und Einsen invertieren diesen. Invertieren heißt: Eine Null wird zu einer Eins und eine Eins zu einer Null. Wenn Sie sich an negative Zahlen in Zweierkomplement-Darstellung erinnern, wissen Sie, daß zur Umwandlung einer positiven Zahl in eine negative und umgekehrt von der Zahl zuerst das Komplement gebildet werden muß. Dies ist nichts anderes als eine Invertierung der einzelnen Ziffern. Für einen Vorzeichenwechsel muß jetzt noch eins zum Komplement addiert werden. Im Programm sieht die gesamte Aufgabe wie folgt aus.

```
ZAHL        =       250         ;ZU WANDELNDE ZAHL
ERG         =       251         ;ERGEBNIS
;
            LDA     ZAHL        ;DIE ZAHL IN DEN AKKUMULATOR LADEN
            EOR     #255        ;ALLE 8 BITS INVERTIEREN
            CLC                 ;EINEN ÜBERTRAG LOESCHEN
            ADC     #1          ;DAS ZWEIERKOMPLEMENT BILDEN
            STA     ERG         ;DAS ERGEBNIS SPEICHERN
            RTS                 ;DAS PROGRAMM VERLASSEN
```

Das Beispiel wandelt eine Zahl entsprechend der Festlegung für das Zweierkomplement in eine positive bzw. negative um.

Aufgabe 7.1 Was erhält man, wenn die Zahl 53 mit 172 UND-, mit 65 ODER-, mit 133 EXOR- und wieder mit 77 UND- verknüpft wird. Variieren Sie die Reihenfolge der Operationen und stellen Sie fest, ob sich das Ergebnis ändert.

Aufgabe 7.2 Versuchen Sie, eine 8-Bit-Subtraktion mit dem Befehl ADC aufzubauen. Nutzen Sie dabei die mathematische Umformung A – B = A + (–B) aus.

Der AND-Befehl wird auch zum Überprüfen des Wertes eines Bits verwendet. In der UND-Maske befindet sich an der Stelle des gewünschten Bits eine Eins. Ist im Datenbyte das Bit ebenfalls gesetzt, erhält man einen Wert ungleich null, ist es jedoch gelöscht, null.

```
    10010101        10010101        10010101
UND 00010000    UND 00000010    UND 10000000
--------------  --------------  --------------
    00010000        00000000        10000000
```

Das Endergebnis kann mit den Befehlen BNE und BEQ leicht überprüft werden und für Programmverzweigungen genutzt werden.

Für derartige Aufgaben gibt es im Befehlssatz des 6502- bzw. 8502-Prozessors eine leichte Abwandlung des AND-Befehls.

BIT prüft Bits

BIT – BIT-test with accumulator
– prüfe den Inhalt des Akkumulators

Der Befehl führt zwischen dem Inhalt des Akkumulators und einer Speicherzelle eine logische UND-Verknüpfung durch. Im Gegensatz zum AND-Befehl werden nur die Statusflags entsprechend dem Ergebnis gesetzt. Aber auch hier gibt es eine Besonderheit.

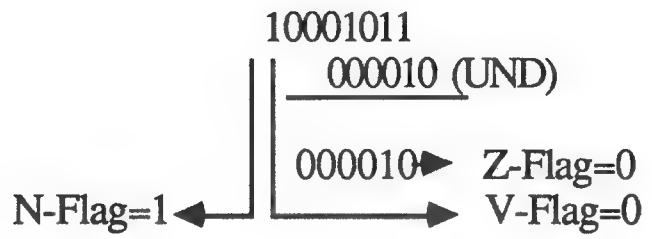
Das Null-Flag zeigt das korrekte Ergebnis der Verknüpfung an. Das Negativ-Flag enthält eine Kopie des Bit 7 vom Inhalt der Speicherzelle. Das Überlauf-Flag enthält entsprechend die Kopie von Bit 6. Der gesamte Sachverhalt in einem Beispiel:

Speicherzelle 250: `10001011`

Programm:

```
LDA  #2
BIT  250
```

Ergebnis:



Das sonderbare Verhalten von N- und V-Flag stellt das Hauptanwendungsgebiet des Befehls dar. Es kann schnell geprüft werden, ob Bit 7 gesetzt ist und damit der Inhalt einer Speicherzelle einen negativen Wert besitzt. Entsprechend kann Bit 6 schnell geprüft werden.

Eine Speicherzelle dient z.B. als Fehlerflag. Tritt ein Fehler auf, wird Bit 7 der Speicherzelle gesetzt. Andere Programmteile können mit dem BIT-Befehl Bit 7 prüfen und entsprechend auf einen Fehler reagieren.

```
BSOUT  =    $FFD2
SUM1        =       250         ;1. SUMMAND
SUM2        =       251         ;2. SUMMAND
ERG         =       252         ;ERGEBNIS
;
            LDA     SUM1        ;DEN 1. SUMMANDEN LADEN
            CLC                 ;DEN UEBERTRAG LOESCHEN
            ADC     SUM2        ;DEN 2. SUMMANDEN ADDIEREN
            STA     ERG         ;DAS ERGEBNIS SPEICHERN
;
            LDA     #"P"        ;POSITIVES ERGEBNIS AUSGEBEN
            BIT     ERG         ;PRUEFEN, OB ERG NEGATIV IST
            BPL     AUSGABE     ;NEGATIV ?, NEIN =>
            LDA     #"N"        ;AKKUMULATOR MIT "N" LADEN
AUSGABE     JMP     BSOUT       ;ZEICHEN AUSGEBEN UND PROGRAMM VERLASSEN
```

Im Beispiel wird eine Ihnen wohlbekannte Addition durchgeführt. Bei einem negativen Ergebnis erscheint das Zeichen "N" und bei einem positiven das Zeichen "P". Nach der Addition nimmt das Programm an, daß das Ergebnis positiv ist und lädt entsprechend den ASCII-Code von "P" in den Akkumulator. Durch den BIT-Befehl wird erst jetzt das Ergebnis geprüft. Der BIT-Befehl wird verwendet, da er den Inhalt des Akkumulators nicht verändert, sondern das Ergebnis nur im Statusregister anzeigt. Ist das Ergebnis positiv, wird direkt zur Ausgabe gesprungen. Ist es jedoch negativ, wird der ASCII-Code von "N" in den Akkumulator geladen und dann ausgegeben. Derartige Programmstrukturen werden Sie häufig in Assemblerprogrammen finden.

Zum einen werden die Vorteile des BIT- Befehls ausgenutzt, zum anderen ist eine Verzweigung in dieser Form kürzer zu programmieren. Hier zum Vergleich die Problemlösung mit einer anderen Programmstruktur:

```
            BIT   ERG
            BPL   POSITIV
            LDA   #"P"
            JMP   AUSGABE
POSITIV     LDA   #"N"
AUSGABE     ...
```

Ein ganz besonderer Befehl

Dieser Befehl gehört eigentlich gar nicht in die Gruppe der Bit-Manipulationen. Aber an anderer Stelle ist er genauso unpassend, darum wird er hier zum Schluß des Kapitels aufgeführt.

NOP – No OPeration
– Keine Operation

Bei diesem Befehl macht der Prozessor überhaupt nichts. Kein Register, keine Speicherzelle und kein Statusflag wird verändert. In Assemblerprogrammen hat der Befehl auch wenig Sinn. Wird ein Maschinenprogramm jedoch ohne einen komfortablen Assembler eingegeben, können mit diesem Befehl Lücken ins Programm eingebaut werden. Diese können dann später mit Befehlen, z.B. Sprüngen zu Testroutinen, ausgefüllt werden.

In Kapitel 11 wird die Funktion eines Monitors erklärt, unter anderem auch der Assembler des Monitors. Bei der Erstellung von Programmen mit dieser Monitorfunktion wird Ihnen die Bedeutung des NOP-Befehls deutlich.

Kapitel 8

Zwei letzte Gruppen von Befehlen

Der Stapel (Stack)

Diese Einrichtung des Prozessors ist wirklich mit einem Stapel zu vergleichen, z.B. mit einem Papierstapel auf Ihrem Schreibtisch. Benötigen Sie ein Blatt Papier gerade nicht, legen Sie es auf den Stapel und holen es dort später wieder herunter. Solche Blätter heften Sie nicht in einen Ordner, da sie wieder schnell zugänglich sein müssen. Genauso legt der Prozessor Daten, die er schnell zwischenspeichern muß, auf den Stapel und holt diese wieder, wenn sie benötigt werden.

Noch eine Eigenschaft hat der Stapel des Prozessors mit einem Papierstapel auf dem Schreibtisch gemeinsam. Werden mehrere Blatt Papier auf ihn gelegt, kann nur das oberste heruntergenommen werden. Um tieferliegende zu erhalten, müssen erst die obersten weggenommen werden. Ebenso muß der Prozessor, um tieferliegende Daten zu entnehmen, die davorliegenden auslesen. Aufgrund dieser Arbeitsweise wird der Stapel "last in, first out oder kurz LIFO-Speicher" bezeichnet. Das zuletzt abgelegte Datum muß auch als erstes wieder gelesen werden.

Der Prozessor kennt vier Befehle, um Daten auf den Stapel zu schreiben bzw. zu lesen.

PHA – PusH Accumulator onto stack
– lege den Akkumulatorinhalt auf den Stapel

PLA – PuLl Accumulator from stack
– hole den Akkumulatorinhalt vom Stapel

PHP – PusH statusregister (P) onto stack
– lege den Inhalt des Statusregisters auf den Stapel

PLP – PuLl statusregister (P) from stack
– hole den Inhalt des Statusregisters vom Stapel

Die ersten zwei Befehle speichern den Akkumualtorinhalt zwischen und die letzten zwei das Statusregister. Diese zwei sind übrigens die einzigen Befehle, die auf das gesamte Statusregister auf einmal zugreifen. Alle bisherigen Befehle, wie CLC oder SED, veränderten nur ein einzelnes Bit.

Die zwei Ablagebefehle wirken natürlich nur als Kopierbefehle und verändern den Inhalt der Register nicht.

Als Anwendungsbeispiel folgt ein Programm, das die zwei Ziffern einer BCD-Zahl in der Speicherzelle 250 vertauscht und das Ergebnis wieder in die Speicherzelle 250 schreibt. Zur Zwischenspeicherung von Werten wird der Stapel verwendet.

```
BCDZAHL     =   250         ;SPEICHERZELLE DER BCD-ZAHL
;
        LDA   BCDZAHL       ;DIE BCD-ZAHL IN DEN AKKU. LADEN
        PHA                 ;DIE ZAHL AUF DEN STACK LEGEN
        LSR   A             ;OBERE ZIFFER NACH RECHTS SCHIEBEN
        LSR   A
        LSR   A
        LSR   A
        STA   BCDZAHL       ;DIE ERSTE ZIFFER MERKEN
;
        PLA                 ;DIE ZAHL WIEDER VOM STACK HOLEN
        ASL   A             ;UNTERE ZIFFER NACH LINKS SCHIEBEN
        ASL   A
        ASL   A
        ASL   A
        ORA   BCDZAHL       ;MIT DER ANDEREN ZIFFER VERKNUEPFEN
        STA   BCDZAHL       ;DAS ENDERGEBNIS SPEICHERN
        RTS                 ;DAS PROGRAMM VERLASSEN
```

Wie die einzelnen Ziffern durch Schiebeoperationen auf die jeweils andere Position gebracht werden, ist aus den letzten Kapiteln klar. Dabei wird jedoch die jeweils andere Ziffer zerstört.

Die Ausgangszahl muß also zwischengespeichert werden. Der Stack erfüllt hier gute Dienste.

Die Zahl wird mit PHA gespeichert, die obere Ziffer verschoben, dann mit PLA die Zahl zurückgeholt und die untere Ziffer verschoben. Die zwei Zwischenergebnisse werden mit ORA verknüpft.

Den zwei Befehlen für das Statusregister kommt in Programmen die Aufgabe zu, die Zustände der Flags zu speichern. Z.B. verändert die Ausgaberoutine BSOUT die Flags. Sollen diese länger erhalten bleiben, muß das Statusregister gespeichert werden.

```
BSOUT    =      $FFD2
SUM1     =      250              ;1. SUMMAND
SUM2     =      251              ;2. SUMMAND
ERG      =      252              ;ERGEBNIS
;
         LDA    SUM1             ;DEN 1. SUMMANDEN LADEN
         CLC                     ;EINEN UEBERTRAG LOESCHEN
         ADC    SUM2             ;DEN 2. SUMMANDEN ADDIEREN
         STA    ERG              ;DAS ERGEBNIS MERKEN
         PHP                     ;ALLE STATUSFLAGS MERKEN
         BCC    PRUEFEV          ;EIN UEBERTRAG ?, NEIN =>
         LDA    #"C"             ;ASCII-CODE VON "C" IN DEN AKKU.
         JSR    BSOUT            ;DIESES ZEICHEN AUSGEBEN
PRUEFEV  PLP                     ;DIE STATUSFLAGS WIEDER HOLEN
         BVC    ENDE             ;EIN UEBERLAUF ?, NEIN =>
         LDA    #"V"             ;ASSCII-CODE VON "V" IN DEN AKKU.
         JSR    BSOUT            ;DIESES ZEICHEN AUSGEBEN
ENDE             RTS            ;DAS PROGRAMM VERLASSEN
```

Das Programm führt eine Addition durch und gibt den Zustand des Übertrag- und Überlauf-Flags aus. Ohne die Zwischenspeicherung der Flags mit PHP würde deren Inhalt durch den Aufruf der BSOUT-Routine zerstört.

Die Inhalte der weiteren Register X und Y können nicht direkt auf dem Stack gesichert werden. Die Inhalte müssen zuerst mit Transfer-Befehlen in den Akkumulator kopiert und dann gespeichert werden. Folgender Programmausschnitt rettet alle Prozessorregister und liest sie wieder aus:

```
PHA                      ;AKKUMULATOR SPEICHERN
TXA                      ;INHALT DES X-REGISTERS IN AKKU.
PHA                      ;DIESEN WERT SPEICHERN
TYA                      ;INHALT DES Y-REGISTERS IN AKKU.
PHA                      ;DIESEN WERT SPEICHERN
...
...
PLA                      ;DEN INHALT DES Y-REGISTERS HOLEN
TAY                      ;DIESEN WERT IN DAS Y-REGISTER
PLA                      ;DEN INHALT DES X-REGISTERS HOLEN
TAX                      ;DIESEN WERT IN DAS X-REGSITER
PLA                      ;DEN INHALT DES AKKU. HOLEN
```

Hier sehen Sie auch, wie wichtig es ist, sich zu merken, in welcher Reihenfolge die Daten gespeichert wurden. Gelesen werden müssen sie in umgekehrter Reihenfolge, wie sie gespeichert wurden. Auch ist es wichtig, daß keine Daten auf dem Stapel zurückbleiben, bevor ein Programm z.B. mit RTS verlassen wird.

Aufgabe 8.1 Schreiben Sie ein Programm, das den Inhalt des Statusregisters in der Speicherzelle 250 ablegt. Dies ist nur über den Stapel möglich.

Die genannten Informationen für den äußerlichen Stapelablauf reichen für die meisten Programme aus. Aber auch die internen Abläufe bei der Stapelverwaltung sind wichtig. Im Arbeitsspeicher des Computers sind die Speicherzellen 256 - 511, also 256 Bytes, zur Aufnahme der Stapeldaten reserviert.

Der Prozessor beginnt ab der Speicherstelle 511 die Daten zu niedrigeren Adressen hin abzulegen. Um wieder zu wissen, an welcher Adresse das letzte Datenbyte steht bzw. an welche das nächste zu schreiben ist, besitzt der Prozessor ein eigenes Register, den Stapelzeiger oder englisch Stackpointer (SP). Der Zeiger ist 8 Bit breit. Aus der Lage der eigentlichen Speicherzellen erkennen Sie dann, daß der Prozessor 256 zum Wert des Stackpointers addiert, was einem fehlenden HI-Byte mit dem Wert 1 entspricht (1 * 256 + 0 = 256, 1 * 256 + 255 = 511).

Der Zeiger teilt dem Prozessor mit, an welche Speicheradresse das nächste Datenbyte geschrieben werden muß. Nach einem Schreibvorgang vermindert der Prozessor den Stapelzeiger automatisch um eins. Zum Lesen des letzten Bytes wird der Stapelzeiger vom Prozessor wieder um ein Byte erhöht und dann das Byte ausgelesen.

Die Vorgänge des Speicherns und Lesens sehen dann wie folgt aus: Der Akkumulator enthält den Wert 101, und der Stapelzeiger zeigt auf die Adresse 503.

```
      501: ---                501: ---                501: ---
      502: ---          SP→   502: ---                502: ---
SP→   503: --- == PHA ⇒       503: 101 == PLA ⇒ SP→   503: ---
      504: 234                504: 234                504: 234
```

Mit dem PHA-Befehl schrieb der Prozessor den Inhalt des Akkumulators in die Speicherzelle 503. Um einen evtl. weiteren Schreibvorgang vorzubereiten, wird der Stapelzeiger um eins vermindert. Beim Lesen durch PLA wird er wieder erhöht, und das Datenbyte kann ausgelesen werden.

Bisher wurde der Prozessor nur durch Befehle des Programms veranlaßt, Daten auf dem Stack abzulegen. Aber auch Informationen für die eigene Verwaltung werden dort gespeichert.

Beim Aufruf von Unterprogrammen mit dem Befehl JSR muß sich der Prozessor den aktuellen Wert des Programmzeigers merken, damit er in das aufrufende Programm mit RTS zurückkehren kann. Der Programmzeiger wird – Sie können es sich sicher schon denken – auf dem Stapel gespeichert. Der 16-Bit-Wert wird, gespalten in LO- und HI-Byte, abgelegt. Zuerst wird das HI-Byte und dann das LO-Byte auf den Stapel gesichert.

```
      501: ---                          SP→  501:  ---
      502: --- == PC: $1317  ⇒               502:   25 ($19)
SP→   503: --- == JSR $FFD2 ⇒               503:   19 ($13)
      504: 234                               504:  234
```

An den Zahlenwerten erkennen Sie, daß der Prozessor nicht die Anfangsadresse des JSR-Befehls ($1317), sondern die des dritten Befehlsbytes ($1319) speichert. Dies ist auch verständlich, denn dies ist die letzte Befehlsinformation, die der Prozessor aus dem Speicher geholt hat.

Der RTS-Befehl liest den alten Wert des Programmzeigers, die sog. Rücksprungadresse, vom Stapel und weist sie wieder dem Programmzähler zu. Bevor der nächste Befehl ausgeführt wird, erhöht der Prozessor die Adresse des Programmzählers um eins.

```
SP→  501:  ---                                  501: ---
     502:   25  ($19)                           502: ---
     503:   19  ($13) == RTS ⇒          SP→     503: ---
     504: 234                                   504: 234
```

Durch die Ablage der Rücksprungadresse auf dem Stapel dürfen Unterprogramme beliebig tief geschachtelt werden. Hier ein Beispielprogramm, das einfach ein "A" auf dem Bildschirm darstellt.

```
BSOUT      =    $FFD2
           JSR  AUSGABE
           RTS
;
AUSGABE    LDA  #"A"
           JSR  BSOUT
           RTS
```

Die grafische Darstellung der Vorgänge auf dem Stack sieht dann wie folgt aus: Es wird angenommen, daß das Programm wie üblich an der Adresse $1300 beginnt.

```
      489: ---                  489: ---                SP→  489: ---
      490: ---                  490: ---                     490:  8
      491: ---                SP→ 491: ---                   491:  19
      492: ---                     492:  2                   492:  2
SP→   493: --- == JSR ⇒           493:  19 == JSR ⇒         493:  19
      494: !GO  AUSGABE          494: !GO AUSGABE          494: !GO
      495: !GO                   495: !GO                  495: !GO
```

```
                    489: —                          489: ---
                    490: ---                        490: ---
                SP→ 491: ---                        491: ---
                    492:  2                         492: ---
== RTS ==>          493:  19 == RTS ⇒           SP→ 493: ---
bei BSOUT           494: !GO bei AUSGABE            494: !GO
                    495: !GO                        495: !GO
```

```
                    489: ---
                    490: ---
                    491: ---
                    492: ---
 == RTS ⇒           493: ---
Programmende        494: ---
                SP→ 495: ---
```

Da der EDASS-Befehl !GO jedes Programm wie ein Unterprogramm aufruft, befindet sich natürlich die Rücksprungadresse zum GO-Programm auch auf dem Stapel. Mit dem letzten RTS-Befehl des Programms wird diese in den Programmzeiger zurückgeladen.

Das Beispielprogramm könnte auch verkürzt werden. Das Unterprogramm zur Ausgabe des Buchstabens "A" könnte die Routine BSOUT auch mit einem JMP-Befehl aufrufen. Der RTS-Befehl am Ende des Unterprogramms entfällt dann. Beim Aufruf der ROM-Routine BSOUT würde jetzt keine Rücksprungadresse auf den Stack gelegt werden.

Der RTS-Befehl am Ende der BSOUT-Routine findet dann auch auf dem Stack keine Rücksprungadresse in das Unterprogramm, sondern eine Adresse, die auf das Hauptprogramm verweist. Diese Methode des Unterprogrammaufrufs (hier BSOUT) findet man sehr häufig. Es wird damit ein RTS-Befehl eingespart.

Dem RTS-Befehl ist es egal, ob die Werte, die er liest, tatsächlich von einem Unterprogrammaufruf mit JSR stammen. Diese könnten genausogut von einem Programm auf den Stapel geschrieben sein. Jetzt wird auch deutlich, warum keine Daten auf dem Stapel vergessen werden dürfen. Ein folgender RTS-Befehl würde diese für den alten Inhalt des Programmzählers halten.

Wie auf den Stapelzeiger direkt zugegriffen wird, ist bis jetzt noch nicht geklärt. Zwei Transferbefehle kopieren dessen Inhalt in das X-Register bzw. den Inhalt des X-Registers in den Stapelzeiger.

TSX – Transfer from Stack-pointer to X-register
– kopiere den Inhalt des Stapelzeigers ins X-Register

TXS – Transfer from X-register to Stack-pointer
– kopiere den Inhalt des X-Registers in den Stapelzeiger

In Assemblerprogrammen werden beide Befehle nur sehr selten benötigt. Das Betriebssystem des C128 setzt den Stapelzeiger nach dem Einschalten auf einen Startwert. Hier dennoch wieder ein Anwendungsbeispiel:

```
TSX             ;INHALT DES STAPELZEIGERS IN X-REG.
LDA 257,X       ;DAS LETZTE DATENBYTE LESEN
```

Durch diese zwei Befehle wird der Inhalt des letzten abgelegten Datenbytes gelesen, ohne den Stapelzeiger zu verändern und damit den letzten Wert vom Stapel zu entfernen.

Unterbrechungen (Interrupts)

Übernimmt der Computer Steuerungsaufgaben, muß das entsprechende Programm schnell auf Ausnahmesituationen reagieren können. Bei der Überwachung eines Heizlüfters z.B. muß die Stromzufuhr bei Überschreiten einer Maximaltemperatur sofort abgeschaltet werden. Müßte das Programm die Temperatur ständig abfragen, hätte es keine Zeit mehr für andere Aufgaben, wie beispielsweise die Ausgabe der aktuellen Temperatur auf dem Bildschirm.

Viel sinnvoller wäre es, wenn elektronische Bausteine dem Prozessor das Eintreten eines Ereignisses mitteilen, dieser das laufende Programm unterbricht und in ein spezielles Programm verzweigt. Im Beispiel könnte dieses z.B. die Stromzufuhr abschalten und eine Meldung für den Benutzer ausgeben.

Der 6502- bzw. 8502-Prozessor besitzt einen Signaleingang, über den ihm Unterbrechungsanforderungen mitgeteilt werden. Das laufende Programm wird unterbrochen, und die sogenannte Interrupt-Routine wird aufgerufen. Im C128 wird mit Unterbrechungen die Tastatur abgefragt. Jede Sechzigstel-Sekunde unterbricht der Prozessor seine Arbeit und prüft, ob eine Taste gedrückt wurde.

Oft ist eine Unterbrechung des Programms aber nicht erwünscht. Werden z.B. wichtige Daten empfangen, könnten durch eine Unterbrechung Informationen verloren gehen. Für diesen Zweck besitzt der Prozessor das Interrupt-Flag im Statusregister. Ist dieses gesetzt, wird keine Unterbrechung mehr zugelassen. Erst wenn das Flag gelöscht wird, wird auch wieder bei einer Unterbrechung die Interrupt-Routine aufgerufen.

SEI – SEt Interrupt enable
– sperre eine Unterbrechung

CLI – CLear Interrupt enable
– erlaube eine Unterbrechung

Ist die Unterbrechung erlaubt, liest der Prozessor die Startadresse der Unterbrechungsroutine aus den Speicherzellen 65534 ($FFFE) und 65535 ($FFFF). Dies sind die letzten zwei Bytes im 64-KByte-Adreßraum des Prozessors. Aufgerufen wird die Interrupt-Routine wie ein Unterprogramm. Am Ende der Routine muß der Prozessor wieder ins unterbrochene Programm zurückkehren können. Im Unterschied zum Unterprogrammaufruf wird zusätzlich das Statusregister auf dem Stapel gespeichert. Auch wird als Rücksprungadresse die Startadresse des nächsten Befehls gesichert.

	496: —		SP→ 496:	---
	497: —		497:	Statusregister
	498: ---	== Interrupt ⇒	498:	LO-Byte
SP→	499: —		499:	HI-Byte
	500: 177		500:	177

Zum Verlassen der Interrupt-Routine ist jetzt ein spezieller Befehl erforderlich.

RTI – ReTurn from Interrupt
– kehre von einer Unterbrechung zurück

Neben der behandelten Art von Unterbrechungen gibt es noch eine weitere. In bestimmten Fällen darf die Möglichkeit einer Sperrung der Unterbrechung gar nicht existieren. Es gibt z.B. Netzgeräte, die einen Stromausfall in Form einer Unterbrechungsanforderung melden und dann die Betriebsspannung noch für einige Millisekunden aufrechterhalten.

In dieser Zeit kann ein Programm wichtige Daten retten. Könnte hier die Unterbrechung gesperrt werden, hätte die ganze Einrichtung ihren Sinn verloren. Derartige Interrupts werden NMI-Interrupt genannt. NMI ist die Abkürzung von "Non-Maskable Interrupt". Dieser Interrupt besitzt auch eine eigene Interrupt-Routine. Deren Startadresse steht in den Speicherzellen 65530 ($FFFA) und 65531 ($FFFB). Im C128 wird bei Drücken der Taste RESTORE ein NMI-Interrupt ausgelöst. Zusätzlich werden damit Steuerungsaufgaben für die RS232-Schnittstelle übernommen.Die beiden besprochenen Interrupts werden von externer Hardware erzeugt. Es gibt aber auch einen Befehl, der softwaremäßig eine Unterbrechung auslöst.

BRK – BReaK
– erzwungene Unterbrechnung

Der Prozessor erhöht nach diesem Befehl den Programmzähler um eins, womit dieser auf den Start des nächsten Befehls zeigt. Danach werden wie bei einem normalen Interrupt Statusregister und Programmzähler auf dem Stack gesichert. Zusätzlich wird das Break-Flag (Bit 4) im Statusregister gesetzt. Verzweigt wird in die normale Interruptroutine, deren Startadresse in den Speicherzellen 65534 ($FFFE) und 65535 ($FFFF) steht.

In dieser Routine wird das Break-Flag geprüft, womit die hard- und softwaremäßigen Unterbrechungsquellen unterschieden werden. Im C128 wird bei einem BRK-Befehl der eingebaute Maschinensprache-Monitor aufgerufen.

Zum Schluß noch ein paar Worte zur Interrupt-Programmierung. Interrupt-Routinen erledigen immer folgende vier Aufgaben:

1. Alle Registerinhalte retten.
2. Die Interrupt-Quelle feststellen.
3. Die eigentliche Interrupt-Aufgabe ausführen.
4. Die Registerinhalte wiederherstellen.

Da die Interrupt-Routinen die Register des Prozessors benutzen, müssen sie beim Aufruf der Routinen gerettet werden. Nur so ist gewährleistet, daß nach Rückkehr ins unterbrochene Programm dieses seine Arbeit fortsetzen kann.

Die Interruptprogrammierung ist eine diffizile Angelegenheit. Vor allem eine genaue Kenntnis der Arbeitsweise der Interrupterzeugenden Chips (Ein-/Ausgabe-Bausteine, Video-Chips, ...) ist erforderlich. Interrupts lassen sich z.B. einsetzen, um auf die Kollision zweier Sprites oder auf das Anliegen eines Signals am User-Port zu reagieren.

Kapitel 9

Wichtiges zur Assembler-programmierung und zum C128

Tips zur Fehlersuche in einem Assemblerprogramm

Die Fehlersuche in einem Assemblerprogramm ist eine sehr schwierige Aufgabe. Formale Eingabefehler können vom Assembler erkannt werden, aber logische Fehler im Programmaufbau zeigen sich erst bei dessen Ablauf. Das Maschinenprogramm gibt nicht wie ein BASIC-Programm eine Fehlermeldung aus, sondern es produziert irgendeinen groben Unsinn. Für den Programmierer gibt es zunächst drei Reaktionsmöglichkeiten des Programms:

1. Das Programm läuft einwandfrei.
2. Das Programm läuft, jedoch ist das Ergebnis nicht ganz korrekt.
3. Das Programm läuft überhaupt nicht oder nur kurz, und der Computer "stürzt" dabei ab.

Zeigen sich nur kleine Funktionsfehler, wie z.B. die Ausgabe von 10 Sternen statt nur 9 oder die falsche Farbe für die Schrift, lassen sich diese schnell beheben. Die Fehlerquelle ist sofort aus der Fehlfunktion erkennbar. Nach einiger Programmier-Erfahrung, vor allem auch in der Fehlersuche, erkennt man auch gröbere Fehler am fehlerhaften Ergebnis.

Was ist aber, wenn alles drunter und drüber läuft, wenn man keine Verbindung zum eingegebenen Programm sieht und der Computer sogar abstürzt? Hier muß die Fehlerquelle Stück für Stück eingegrenzt werden. Meist läuft das Programm eine kurze Zeit und trifft erst dann auf den Fehler. Eine grobe Abgrenzung kann damit bereits vorgenommen werden. Meist tritt ein Fehler nach einer Programmänderung auf.

Es ist also sehr wahrscheinlich, daß im neu eingegebenen Programmtext ein Fehler sitzt. Vielleicht stören sich neue und alte Programmteile. Wenn damit der Fehler nicht gefunden wird, müssen Werte über den Programmablauf gesammelt werden. Oft schreibt das Programm Zwischenergebnisse in den Speicher. Man kann z.B. prüfen, ob diese Werte bereits abgelegt wurden und ob die Werte stimmen. Damit läßt sich eine Aussage gewinnen, wie weit das Programm vor dem Fehler bearbeitet wurde.

Sollten die vom Programm gespeicherten Zwischenwerte nicht ausreichen, fügen Sie zur Fehlersuche einfach einige Speicherbefehle (STA, STX, STY) zusätzlich in das Programm ein. Schleifenzähler können damit z.B. beobachtet werden.

Sollte der Fehler jetzt noch immer nicht gefunden sein, gibt es noch eine durchgreifende Methode zur Lokalisierung des Fehlers. Der Assemblerbefehl BRK unterbricht das laufende Programm und verzweigt in den Maschinensprache-Monitor. Bauen Sie diesen Befehl ins Programm ein, und setzen Sie ihn von Programmablauf zu Programmablauf an eine andere Stelle. Zum einen können Sie mit dem Monitor Speicherwerte schnell überprüfen, zum anderen sehen Sie an der Tatsache, daß der Monitor aufgerufen wurde, wie weit das Programm abgearbeitet wurde. Übrigens noch ein Wort zum Aufruf des Monitors:

Sollte dieser bei einem fehlerhaften Programmablauf ohne einen bewußt eingebauten BRK-Befehl aufgerufen worden sein, ist der Programmzähler außerhalb des Programms geraten. Im Datenspeicher des C128 stehen unterschiedliche Werte, unter anderem auch Nullen. Dies entspricht dem Objektcode des BRK-Befehls. Ein unbeabsichtigter Aufruf des Monitors ist also auch ein Hinweis auf einen schweren Fehler.

Jetzt soll das Aussehen häufiger Fehler beschrieben werden. Eine Ursache sind Tippfehler. Schnell wird aus "LDA" "LDX", aus "STY" "STX" oder aus "TAX" "TXA". Auch ein falsch eingegebener Labelname kann böse Folgen haben. Gerade bei Fehlern, die nach neu eingegebenem Text auftreten, sollten Sie diese Ursache überprüfen. Unter diese Gruppe der Fehler fällt auch das Verwechseln zweier Befehle. So hat der Befehl PHA statt PLA fatale Folgen.

Liegt kein Tippfehler vor, ist es schwer, allgemeingültige Hinweise zu geben. Beliebte Fehlerquellen sind z.B. falsch gesetzte Flags. Die genaue Kenntnis deren Beinflussung und Auswirkung ist deshalb unbedingt nötig. In Zusammenhang mit indizierter Adressierung können falsche Werte des X- und Y-Registers Fehler produzieren. Bei der nach-indizierten indirekten Adressierung kommt die Steuerung des Zeigers in der Zero-Page hinzu.

Eine Fehlerquelle mit ziemlich wüsten Folgen stellen die Stapel-Befehle dar. Wird vergessen, Werte vom Stapel zu entfernen oder werden zu viele gelesen, findet ein nachfolgender RTS-Befehl auf dem Stapel nicht mehr die Rücksprungadresse zum aufrufenden Programm. Bei Verzweigungen mit den Branch-Befehlen kann es leicht passieren, daß Stapelzugriffe übersprungen werden, die den Stapelzeiger verändern. Das Programm muß so aufgebaut sein, daß egal, welcher Weg bei einer Verzweigung gewählt wird, am Ende der Stapelzeiger auf dieselbe Adresse zeigt.

Zum Schluß eine Fehlerquelle, die für den Commodore 128 ganz typisch ist. Wird im Programm zwischen verschiedenen Speicherbereichen umgeschaltet, kann es leicht zu Fehlern kommen. Danach stürzt der Computer meist ab. Näheres zur Speicherverwaltung erfahren Sie im entsprechenden Abschnitt dieses Kapitels.

Ist der Fehler endlich lokalisiert, muß immer bedacht werden, daß die Programmstelle, an der der Fehler aufgetreten ist, und die Programmposition, die ihn verursacht hat, nicht immer identisch sind. Bei der ganzen Fehlersuche ist eine Kommentierung des Programms sehr hilfreich. Der Kommentar sollte die Funktion der Programmzeile oder eines kurzen Programmabschnitts beschreiben. Beim Analysieren des Programms sind Sie dann nicht auf die Befehlswörter allein angewiesen.

Noch mehr Speicher durch Banking

Alle technischen Informationen zur Speicherverwaltung im C128 und dem Prinzip des Banking finden Sie im Handbuch Ihres C128 in Anhang B. Hier soll nur eine Begriffsklärung und Einführung gegeben werden.

Der Prozessor 8502 besitzt einen sogenannten 16 Bit breiten Adreßbus. Liest der Prozessor den Inhalt einer Speicherzelle, wird über diesen die gewünschte Adresse dem Speicherbaustein mitgeteilt. Mit 16 Bits lassen sich die Zahlen von 0 bis 65535 darstellen. Dies entspricht einem ansprechbaren Speicher von 64 KByte (1Kbyte = 1024 Bytes). Einen größeren Speicher kann der Prozessor nicht adressieren.

Um diese Begrenzung zu umgehen, baut man in den Computer elektronische Schalter ein, die z.B. zwischen zwei Speicherbausteinen hin- und herschalten. Der Prozessor spricht weiterhin nur 64 KByte an. Steht der Schalter jedoch vor dem Zugriff auf Speicherbereich 1, wird auf diesen zugegriffen. Danach kann auf den zweiten Speicherbereich umgeschaltet werden, und derselbe Befehl führt dann einen Zugriff auf Speicherbereich 2 aus. Die im Adreßraum parallel liegenden Speicherbereiche werden "Banks" genannt. Die Methode dieser Speichererweiterung entsprechend "Banking".

Zwei Eigenschaften dieser Methode werden jetzt auch deutlich. Der gleiche Befehl mit der gleichen Speicheradresse schreibt z.B. nach dem Umschalten der Speicherbank in eine andere Speicherzelle. Am Befehl selbst ist dies nicht zu erkennen. Die zweite Neuerung ist, daß das Programm selbst, sozusagen von "Hand", zwischen den Speicherbanks umschalten muß. Eine intensive Nutzung des gesamten Speichers erfordert viele Umschaltungen. Diese kosten Zeit und machen ein Programm langsamer. Auch ist eine genaue Kontrolle der

Umschaltvorgänge wichtig. Es ist sonst schnell passiert, daß das Programm den Ast absägt, auf dem es selbst sitzt. Genauer müßte man sagen, daß es die Speicherbank abschaltet, in der es sich selbst befindet. Zwar arbeitet der Prozessor danach korrekt weiter, aber im neuen Speicherbereich, aus dem jetzt das Programm gelesen wird, befinden sich nur sinnlose Werte, die ein ebenfalls sinnloses Programm bilden.

Wie ist jetzt der gesamte Speicher des C128 aufgebaut? In der Grundversion besitzt der Computer 2 RAM-Speicherbänke (Bank 0 und Bank 1). Beide umfassen 64 KByte, womit insgesamt 128 KByte Speicher entstehen. Durch Speichererweiterungen kann dieser jedoch bis auf 1 MByte ausgebaut werden. Diese sind dann natürlich auch wieder in 64 KByte Bänke unterteilt.

Der Festwertspeicher des C128 ist auch in Banks aufgeteilt. Die normalen ROMs, wie BASIC-Interpreter und Betriebssystem, liegen in einer Bank. Die am Erweiterungsport zusteckbaren ROMs, wie Spiele oder Programmiersprachen, befinden sich in einer weiteren Bank.

Wie werden die einzelnen Banks gewählt? Ein spezieller Chip, die sog. MMU (Memory Management Unit), übernimmt die gesamte Verwaltung. Über bestimmte Speicherzellen wird ihr die gewünschte Speicherbank mitgeteilt. Es wird sofort der gesamte 64-kByte-Adreßraum umgeschaltet, so daß beim nächsten Befehl der richtige Speicher vorliegt. Die ROMs haben eine Sonderstellung.

Diese werden abschnittsweise zum normalen Speicher hinzugeschaltet. Es ist z.B. möglich, Bank 0 zu wählen und in einem begrenzten Teil das Betriebssystem einzublenden. Genauso wäre es möglich, den BASIC-Interpreter oder den Zeichensatz hinzuzuschalten. Dieser gleichzeitige Betrieb von RAM- und ROM-Speicher ist wichtig. Wie sollte denn sonst ein Programm im RAM-Speicher eine Routine des Betriebssystems im ROM-Speicher aufrufen? In der folgenden Tabelle sind die Adreßlagen der einzelnen ROM-Abschnitte eingetragen:

$0000 – $4000	immer RAM-Speicher der aktuellen Bank
$4000 – $7FFF	erster Teil des BASIC-Interpreters
$8000 – $BFFF	zweiter Teil des BASIC-Interpreters und Monitor
$C000 – $FFFF	das Betriebssystem

Es gibt noch ein zweites Problem. Aus dem Kapitel über Adressierarten wissen Sie, daß den ersten 256 Bytes, der sog. Zero-Page, eine besondere Aufgabe zukommt. Auch der Bereich 256 bis 511, in dem sich der Stapel des Prozessors befindet, ist für den Programmablauf lebenswichtig. Beim Umschalten der Speicherbank, was immer die vollen 64 KByte betrifft, würden auch

diese Speicherbereiche sich ändern. Dies ist ein in der Regel unerwünschter Effekt. Im C128 wurde das Problem durch eine sogenannte "Common Area" gelöst, was soviel wie "gemeinsamer Speicherbereich" heißt. Im Speicherverwaltungsbaustein kann die Lage und Größe eines Speicherbereichs bestimmt werden, der nicht umschaltbar ist.

Dieser liegt physikalisch immer in Bank 0. Wird jetzt z.B. auf Bank 1 geschaltet, greift der Prozessor bei den markierten Adressen weiterhin auf Bank 0 zu. Im Normalfall sind die ersten 1024 Bytes (0 – $3FF) des Speichers als Common Area definiert. Dieses Verfahren kann nicht auf die ROMs angewendet werden. Diese schalten auch einen gemeinsamen Bereich ab.

Der BASIC-Befehl BANK und die Assembleranweisung .BANK erlauben die Wahl einer Speicherbank. Die Befehle schalten diese nicht sofort um, sondern erst beim nächsten Speicherzugriff, z.B. mit PEEK, SYS oder beim Schreiben des Objektcodes. Die als Parameter angegebene Nummer entspricht nicht exakt der Banknummer, sondern bestimmten interessanten Kombinationen des Speicheraufbaus. Diese lauten wie folgt:

BANK	0	im ganzen Adreßraum Bank 0
	1	im ganzen Adreßraum Bank1
	2	im ganzen Adreßraum Bank2 (mit Speicheerrweiterung)
	3	im ganzen Adreßraum Bank3 (mit Speichererweiterung)
	4	Bank 0 und Internal Function ROM und I/O-Chips
	5	Bank 1 und Internal Function ROM und I/O-Chips
	6	Bank 2 und Internal Function ROM und I/O-Chips
	7	Bank 3 und Internal Function ROM und I/O-Chips
	8	Bank 0 und External Function ROM und I/O-Chips
	9	Bank 1 und External Function ROM und I/O-Chip
	10	Bank 2 und External Function ROM und I/O-Chip
	11	Bank 3 und External Function ROM und I/O-Chip
	12	Bank 0 + Betriebssystem + Internal Function ROM + I/O
	13	Bank 0 + Betriebssystem + External Function ROM + I/O
	14	Bank 0 + Betriebssystem + BASIC-ROMs + Zeichensatz
	15	Bank 0 + Betriebssystem + BASIC-ROMs + I/O-Chips

Die beim Assemblieren gewählte Speicherkombination bleibt bis zu einem späteren Aufruf des Programms mit SYS oder !GO erhalten. Wenn Sie die in den Beispielen geforderte Speicherwahl .BANK 15 mit der Tabelle vergleichen, wird Ihnen auch klar, wieso diese gewählt wurde.

Dies ist die einzige Kombination, in der Bank 0, Betriebssystem, BASIC und die Ein-/Ausgabebausteine gleichzeitig eingeschaltet sind. Bei Ihren ersten eigenen Programmen sollten Sie immer diese Kombination wählen. Wird das

Assemblerprogramm in ein BASIC-Programm eingebaut, müssen Sie vor dem Aufruf des Programms mit SYS die Speicherbank mit BANK 15 wählen.

Als Einsteiger in die Assemblerprogrammierung haben Sie hier eine Erklärung der Arbeitsweise des Bankings erhalten. Den genauen Ablauf der Steuerung lesen Sie bitte im Handbuch des C128 nach. Für viele Programme reicht der freie Speicher in Bank 0 von $1300 – $19FF ($1A00 – $1BFF ist von EDASS belegt) vollkommen aus.

Das Betriebssystem des C128

Ein Betriebssystem hat die Aufgabe, dem Hauptprogramm so grundlegende Arbeiten wie die Ausgabe eines Zeichens auf dem Bildschirm, die Übermittlung von Daten an die Diskette oder die Abfrage der Tastatur abzunehmen. Warum soll sich z.B. ein Textverarbeitungsprogramm darum kümmern, wie ein Zeichen auf dem Bildschirm dargestellt wird?

Für das Programm genügt es, dem Betriebssystem mitzuteilen, welches Zeichen wo auf dem Bildschirm erscheinen soll. Nur so ist es möglich, daß Programme mit minimalem Aufwand die 40- und 80-Zeichen-Darstellung des C128 ansprechen können. Das Betriebssystem spricht je nach gewählter Darstellung die eine oder andere Darstellung an.

Im C128 ist das Betriebssystem nochmals in Editor ($C000 – $CFFF), Unterprogramme zur Darstellung von Zeichen auf dem Bildschirm und zur Texteingabe, und in das eigentliche Betriebssystem ($E000 – $FFFF) mit den Ein-/Ausgaberoutinen aufgeteilt. Jedes Unterprogramm hat eine Startadresse, über die es aufgerufen wird. Den Aufruf der wichtigsten Routinen hat man jedoch in sogenannten Sprungleisten zusammengefaßt.

Diese sind nichts anderes als eine Folge von mehreren JMP-Befehlen, die zu den Unterprogrammanfängen springen. Diese Sprungleisten sollten Sie immer verwenden, sie haben nämlich noch einen weiteren Sinn. Sollte das Betriebssystem geändert werden, würden sich wahrscheinlich auch die Anfänge der Unterprogramme verschieben.

Ältere Programme wären dann nur auf der alten Version des Betriebssystems lauffähig. Die Sprungleiste wird bei Änderungen jedoch bewußt an der gleichen Position belassen.

Es ändern sich nur die Sprungadressen der JMP-Befehle, aber deren Startadressen bleiben bestehen. Programme, die immer die Sprungleiste aufrufen, haben folglich auch keine Schwierigkeiten mit Versionsänderungen. Ein Teil der Sprungleiste (ab $FF81) ist sogar identisch mit der des C64.

Im folgenden erhalten Sie eine Erklärung der Betriebssystem-Routinen, die Lage der übergebenen Parameter und einige kurzer Programmbeispiele. In der Überschrift jedes Unterprogramms steht die Adresse des JMP-Befehls in der Sprungleiste, der Name der Routine und einige Worte zur Erläuterung.

Als erstes zum Editor:

$C000 – CINT – Video-Chips und Editor initialisieren

Diese Routine versetzt die Video-Chips der 40- und 80-Zeichendarstellung in die Ausgangsstellung, wie sie auch nach dem Einschalten des Computers besteht. Außerdem werden alle vom Editor verwendeten Speicherzellen in der Zero-Page initialisiert. Als Darstellung erscheint nach dem Aufruf der Routine die mit der 40/80-Umschalttaste gewählte Anzeigeart.

$C003 – DISPLY – Ausgabe eines Zeichens (AC) mit Attribut (XR)

Mit diesem Unterprogramm kann ein Zeichen direkt auf den Bildschirm geschrieben werden. Die Ausgabeposition ist die aktuelle Cursorposition. Nach der Ausgabe wird diese jedoch nicht nach links versetzt. Das Zeichen selbst wird im Akkumulator übergeben und muß im Commodore-eigenen Bildschirmcode vorliegen. Die Farbe bzw. im 80-Zeichen-Modus zusätzlich die Darstellungsart wird im X-Register übergeben. Eine Tabelle der verschiedenen Bildschirmcodes und Attributwerte finden Sie in Anhang H. Das folgende kurze Beispiel gibt ein "B" auf dem Bildschirm aus, das im 80-Zeichen-Modus zusätzlich blinkt.

```
DISPLY =    $C003
        LDA  #2          ;BILDSCHIRMCODE VON "B"
        LDX  #23         ;FARBE GELB UND BLINKEN
        JMP  DISPLY      ;ZEICHEN AUSGEBEN UND PROGRAMMENDE
```

$C006 – GETKEY – ein Zeichen aus dem Tastaturpuffer holen

Jede Sechzigstel-Sekunde wird die Tastatur über eine Interrupt-Routine abgefragt. Da das Programm die Tastendrücke nicht sofort übernehmen kann, werden sie in einem Speicherabschnitt, dem Tastaturpuffer, abgelegt. Es können dort bis zu zehn Tastencodes gespeichert werden. Bei Aufruf dieser Routine wird der ASCII-Code der Taste aus dem Puffer entnommen und dem Programm im Akkumulator übergeben. Wurde eine Funktionstaste gedrückt, übermittelt diese Routine statt dessen den Text der Funktionstaste. Dieser kann, wie Sie wissen, mit dem BASIC-Befehl KEY geändert werden. Als Beispiel werden die Zeichen der Tastatur ausgelesen und auf dem Bildschirm dar-

gestellt. Es erscheint dabei kein Cursor. Abgebrochen wird das Programm bei der Eingabe eines Ausrufezeichens. Um festzustellen, ob sich Zeichen im Tastaturpuffer befinden, wird die Speicherzelle 208 ausgelesen. Diese enthält die Anzahl der gepufferten Zeichen.

```
GETKEY    =    $C006
PRINT     =    $C00C
SCHLEIFE  LDA  208          ;ZEICHEN IM PUFFER
          BEQ  SCHLEIFE     ;NEIN =>
          JSR  GETKEY       ;ZEICHEN AUS DEM TASTATURPUFFER
          CMP  #"!"         ;MIT AUSRUFEZEICHEN VERGLEICHEN
          BEQ  ENDE         ;IDENTISCH ?, JA =>
          JSR  PRINT        ;DAS ZEICHEN AUSGEBEN
          JMP  SCHLEIFE     ;ZUM ANFANG DER SCHLEIFE
ENDE      RTS               ;DAS PROGRAMM VERLASSEN
```

$C009 – INPUT – Zeichen vom Bildschirm in Akkumulator lesen

Nach dem ersten Aufruf dieses Unterprogramms muß ein Text eingegeben werden. Wie bei der normalen Eingabe von Befehlen, kann der Cursor dabei frei bewegt werden. Nach Drücken der RETURN-Taste wird der Inhalt der aktuellen Cursorzeile im Akkumulator übergeben, bei diesem ersten Aufruf das erste Zeichen, beim zweiten das zweite usw.

Die Zeichen selbst werden in den ASCII-Code umgewandelt. Als letztes Zeichen der Zeile wird ein Wagenrücklauf übergeben.

$C00C – PRINT – ein Zeichen (AC) auf dem Bildschirm ausgeben

Im Gegensatz zu der oben beschriebenen Routine muß hier das Zeichen im ASCII-Code vorliegen. Ein Attribut kann nicht mehr angegeben werden. Jetzt ist es aber möglich, Steuerzeichen wie Cursor links, Bildschirm löschen, Tabulator setzen oder Farbe ändern zu übermitteln. Auch wird nach der Ausgabe eines Zeichens der Cursor nach links gesetzt.

```
PRINT  =     $C00C
       LDA   #"A";          ;ASCII-CODE VON "A" LADEN
       JSR   PRINT          ;DAS ZEICHEN AUSGEBEN
       LDA   #"B"           ;ASCII-CODE VON "B" LADEN
       JMP   PRINT          ;DAS ZEICHEN AUSGEBEN
```

$C00F – SCRORG – Aktuelle Bildschirmgröße lesen

Beim C128 können Teile des Bildschirms als sogenanntes Window definiert werden. Ausgaben mit PRINT sind dann auf diesen Bereich beschränkt. Mit

dieser Routine erfahren Sie die Größe des aktuellen Windows. Als Parameter werden im X-Register die maximal zulässige Anzahl Spalten (Zeichen pro Zeile) und im Y-Register die maximale Anzahl Zeilen zurückgegeben. Zusätzlich wird im Akkumulator die aktuelle Bildschirmbreite, also 40 oder 80 Zeichen, mitgeteilt.

$C018 – PLOT – Cursorposition lesen bzw. schreiben

Die Funktion der Routine wird mit dem Übertrags-Flag (C-Flag) ausgewählt. Wurde das Flag vor dem Routinenaufruf, z.B. mit SEC, gesetzt, wird die aktuelle Cursorposition gelesen. Im X-Register wird die Zeilenposition und im Y-Register die Spaltenposition zurückgegeben. Ist das Flag beim Aufruf gelöscht, wird die Cursorposition neu gesetzt. Entsprechend gibt das X-Register die Zeilen- und das Y-Register die Spaltenposition vor. Sowohl beim Lesen als auch beim Schreiben der Cursorposition sind die Zahlenwerte relativ zur aktuellen linken oberen Ecke des Ausgabefensters berechnet. Ist ein solches nicht definiert, ist die Ecke identisch mit der linken, oberen Bildschirmecke.

Übrigens kann die Cursorposition auch bei nicht sichtbarem Cursor gelesen und geschrieben werden. Die Cursorposition bestimmt immer die nächste Ausgabeposition eines Zeichens. Das folgende Beispiel setzt den Cursor zuerst in die HOME-Position und gibt dann in der zehnten Zeile und der 20. Spalte einen Punkt aus. Bei allen Zahlenangaben ist zu beachten, daß die Zählung bei null beginnt. Für die zehnte Zeile muß also 9 übergeben werden.

```
PRINT  =   $C00C
PLOT   =   $C018
       LDA #29              ;ASCII-CODE VON HOME LADEN
       JSR PRINT            ;DAS ZEICHEN AUSGEBEN
       CLC                  ;C-FLAG LOESCHEN=>CURSOR SETZEN
       LDX #9               ;X-REGISTER=ZEILENPOSITION=9
       LDY #19              ;Y-REGISTER=SPALTENPOSITION=19
       JSR PLOT             ;DEN CURSOR AN DIE NEUE POSITON
       LDA #"."             ;ASCII-CODE VON "." LADEN
       JMP PRINT            ;ZEICHEN AUSGEBEN UND PROGRAMMENDE
```

$C01B – CURSOR – Cursorposition an 80-Video-Chip übermitteln

Auf dem 40-Zeichen-Bildschirm wird der Cursor durch Invertieren eines Zeichens erzeugt. Dies geschieht automatisch in der Interrupt-Routine. Der Video-Chip zur Darstellung der 80 Zeichen erzeugt den Cursor jedoch selbst, so daß nur die aktuelle Cursorpositon an den Chip übermittelt werden muß. Benötigt wird diese Routine nur selten, da das Betriebssystem bei jedem Umsetzen des Cursors dies sofort dem Video-Chip mitteilt.

$C01E – ESCAPE – Bearbeitung einer ESC-Sequenz

Der C128 bietet in seinem Editor ESC-Sequenzen an. Durch aufeinanderfolgendes Drücken der Tasten ESC und einer Buchstabentaste werden die einzelnen Funktionen aufgerufen. Von einem Assemblerprogramm aus müßte über die bereits beschriebene Routine JPRINT ein ESC-Code (27) und dann das entsprechende Folgezeichen gesandt werden. Um diese Prozedur abzukürzen, können die einzelnen Funktionen mit diesem Unterprogramm direkt aufgerufen werden. Zur Auswahl wird nur der Kennbuchstabe der ESC-Sequenz im Akkumulator übergeben. Das Beispielprogramm rollt den Bildschirm zwei Zeilen aufwärts.

```
ESCAPE    =   $C01E
          LDA #"V"            ;ASCII-CODE VON "V" LADEN
          JSR ESCAPE          ;ESC-SEQUENZ MIT DIESEM ZEICHEN
          LDA #"V"            ;ASCII-CODE VON "V" LADEN
          JMP ESCAPE          ;ESC-SEQUENZ MIT DIESEM ZEICHEN
```

$C021 – KEYSET – Funktionstaste belegen

Mit diesem Unterprogramm kann eine Funktionstaste mit einem Text belegt werden. Da die Verwaltung der Tastentexte im Computer kompliziert gestaltet ist, sollten Sie diesen Weg immer wählen, um in einem Maschinenprogramm die Funktionstasten zu belegen. Der Routine wird im Akkumulator die Adresse dreier Informationsbytes aus der Zero-Page übergeben. Im X-Register muß die Nummer der Funktionstaste und im Y-Register die Länge des Textes stehen. Die drei Informationsbytes in der Zero-Page werden am besten in die Speicherzellen 250 – 252 gelegt. Diese enthalten folgende Parameter:

1. Byte (250): LO-Byte der Anfangsadresse des Textes
2. Byte (251): HI-Byte der Anfangsadresse des Textes
3. Byte (252): der Nummer der Bank, in der sich der Text befindet

Die Routine erstellt vom Belegungstext natürlich eine Kopie, so daß der Quellbereich des Textes wieder verwendet werden kann. Als Beispiel wird die Funktionstaste F7 mit dem Text "Funktionstaste" belegt:

```
INFO      =   250
KEYSET    =   $C021
          LDA   #<TEXT          ;LO-BYTE DER ADRESSE DES TEXTES
          STA   INFO            ;SPEICHERN
          LDA   #>TEXT          ;HI-BYTE DER ADRESSE DES TEXTES
          STA   INFO+1          ;SPEICHERN
          LDA#0                 ;BANKNUMMER DES TEXTES
          STA   INFO+2          ;SPEICHERN
```

```
        LDA    #INFO              ;ADRESSE DER INFORMATIONSBYTES
        LDX    #7                 ;NUMMER DER FUNKTIONSTASTE
        LDY    #14                ;LAENGE DES TEXTES
        JMP    KEYSET             ;TASTE BELEGEN +PROGRAMMENDE
TEXT    .TEXT "FUNKTIONSTASTE"    ;BELEGUNGSTEXT
```

Die zwei Tasten RUN/STOP und HELP stellen auch nichts anderes als Funktionstasten dar. Nach dem Einschalten des C128 wird z.B. HELP mit dem Text "HELP" belegt. Im Gegensatz zum BASIC-Befehl KEY kann mit dieser Routine auch diesen zwei Tasten ein Text zugewiesen werden. RUN/STOP besitzt die Nummer 9 und HELP die Nummer 10.

Werden Befehle als Funktionstexte übergeben und sollen diese nach Drücken der Taste ausgeführt werden, muß in den Text ein Wagenrücklauf eingebaut werden (z.B. .TEXT "LIST",13). Bei der Belegung ist zu beachten, daß alle Funktionstexte zusammen nur 246 Bytes lang sein dürfen.

$C02A – SWAPPR – Umschalten zwischen 40-/80-Zeichen-Darstellung

Diese Routine ist in der Funktion identisch mit der Sequenz ESC+X. Beim Aufruf schaltet das Betriebssystem die Ausgabe auf die jeweils andere Darstellung um. Nach zwei Aufrufen ist wieder die alte Darstellungsart aktiviert.

$C02D – WINDOW – Bildschirmfenster neu definieren

Das Unterprogramm definiert die zwei dieagonal gegenüberliegenden Ecken eines Bildschirmfensters neu. Sie ist funktionsgleich mit den zwei Sequenzen ESC+"B" und ESC+"T". Das Übertrags-Flag (C-Flag) wählt die gewünschte Ecke aus. Bei gelöschtem Flag wird die obere, linke Ecke neu definiert und bei gesetztem Flag die untere, rechte Ecke. Im Akkumulator wird die Zeilennummer (0 bis 24) und im X-Register die Spaltenposition (0 bis 39) übergeben. Als Beispiel wird ein 10 *10 Zeichen großes Fenster definiert.

```
WINDOW   =   $C02D
         CLC                  ;C-FLAG GELOESCHT => LINKS OBEN
         LDA   #5             ;AKKUMULATOR=ZEILENPOSITION=5
         LDX   #5             ;X-REGISTER=SPALTENPOSITION=5
         JSR   WINDOW         ;DIE ECKE DEFINIEREN
         SEC                  ;C-FLAG GESETZT => RECHTS UNTEN
         LDA   #14            ;AKKUMULATOR=ZEILENPOSITION=14
         LDX   #14            ;X-REGISTER=SPALTENPOSITION=14
         JMP   WINDOW         ;ECKE DEFINIEREN +PROGRAMMENDE
```

Zwei sehr nützliche Routinen sind in der Sprungleiste leider nicht enthalten: das Ein- und Ausschalten des Cursors.

$CD6F – CRSRON – den Cursor einschalten

Egal, ob der 40- oder 80-Zeichen-Modus aktiv ist, es wird der entsprechende Cursor eingeschaltet. Er erscheint an der durch die letzte Ausgabe bestimmten Position.

$CD9F – CRSROF – den Cursor abschalten

Diese Routine schaltet den Cursor wieder ab. Im Betriebssytem wird seine Position jedoch weiterhin zur Bestimmung der nächsten Ausgabeposition verwendet. Es folgt jetzt die Beschreibung der verschiedenen Aufrufe des Betriebssystems. Dieses wird übrigens mit einem Fachwort auch Kernal genannt.

$FF47 – SPINOT – auf schnellen Floppybetrieb schalten

Diese Routine schaltet die schnelle Datenübertragung zur Floppy ein bzw. aus. Bei gelöschtem Übertrags-Flag (C-Flag) wird diese eingeschaltet und bei gesetztem Flag ausgeschaltet. Die schnelle Übertragung kann natürlich nur mit den Floppystationen 1570 und 1571 genutzt werden.

$FF4A – CLOSAL – alle Dateien eines Gerätes schließen

Mit dieser Routine werden alle Dateien, die zu einem Gerät geöffnet wurden, wieder geschlossen. Die Geräteadresse wird im Akkumulator übergeben. Ist nach Abschluß der Routine das Übertrags-Flag (C-Flag) gesetzt, so ist bei der Bearbeitung ein Fehler aufgetreten.

$FF4D – C64MD – schaltet auf den C64-Modus

Diese Routine schaltet in den C64-Modus. In Assemblerprogrammen wird sie wohl nie benötigt, denn das Überwechseln in einem Programm ergibt wenig Sinn.

$FF53 – BOOTCL – Boot-Programm von Diskette laden und starten

Auf jeder Diskette kann ein sogenannter Boot-Sektor eingetragen werden, in dem sich ein kurzes Programm befindet. Es muß im X-Register die Gerätea-

dresse der Diskettenstation und im Akkumulator der ASCII-Code der Laufwerksnummer (0 = 48 oder 1 = 49) übergeben werden. Die Routine liest danach Sektor 0 in Spur 1. Befindet sich dort ein entsprechender Kenntext, werden aus dem Sektor weitere Informationen, wie die Startadresse des Programms, gelesen. Diesen Boot-Vorgang versucht der C128 bei jedem Einschalten oder beim Drücken der RESET-Taste.

So ist es z.B. möglich, ohne einen Tastendruck gleich beim Start des Computers ein Programm zu laden. Das Beispiel führt einen Bootvorgang von der Dis- kettenstation mit der Nummer 8 durch.

```
BOOTCL    =     $FF53
          LDA   #"0"          ;LAUFWERK 0
          LDX   #8            ;GERAET 8
          JMP   BOOTCL        ;PROGRAMM BOOTEN
```

$FF59 – LKUPLA – Geräte- und Sekundäradresse suchen

Wird im C128 eine Datei geöffnet, muß dem Betriebssystem eine logische Nummer, eine Geräteadresse und eine Sekundäradresse mitgeteilt werden. Die logische Nummer dient zur Unterscheidung der einzelnen Dateien. Die Routine gibt zu einer logischen Nummer, übergeben im Akkumulator, die Geräte- und Sekundäradresse zurück.

Das X-Register enthält die Geräteadresse und das Y-Register die Sekundäradresse. Ist nach Aufruf der Routine das Übertrags-Flag gesetzt, ist die gewünschte Datei nicht geöffnet.

$FF5C – LKUPSA – Geräteadresse über Sekundäradresse suchen

Diese Routine sucht zu einer im Y-Register vorgegebenen Sekundäradresse die dazugehörige Geräteadresse und logische Dateinummer. Wurde keine entsprechende Sekundäradresse gefunden, ist nach Aufruf der Routine das Übertrags-Flag gesetzt. Die Geräteadresse wird im X-Register und die logische Dateinummer im Akkumulator zurückgegeben.

$FF5F – JPSWAP – umschalten zwischen 40-/80-Zeichen-Darstellung

Dieser Sprungbefehl ruft die bereits beschriebene Editor-Routine SWAPPR auf.

$FF65 – JPFKEY – Funktionstaste mit neuem Text belegen

Mit dieser Routine wird die Funktion KEYSET des Editors aufgerufen.

$FF68 – SETBNK – Bank für LOAD/SAVE/VERIFY bestimmen

Für diese drei BASIC-Befehle muß eine Speicherbank bestimmt werden, aus der die Daten gelesen bzw. in die sie geschrieben werden. Im Akkumulator muß die Speicherbank und im X-Register die entsprechende Bit-Kombination für den Speicherverwaltungs-Chip übergeben werden.

$FF6B – GETCFG – MMU-Wert für eine Bank bestimmen

Im letzten Abschnitt über Banking wurden 15 mit dem Befehl BANK wählbare Speicherkombinationen beschrieben. Diese Kombinationen sind in Wirklichkeit im Betriebssystem festgelegt.

Mit dieser Routine wird zu einer gewünschten Bank, deren Nummer im X-Register übergeben wird, die entsprechende Bit-Kombination für den MMU-Baustein (Speicherverwaltung) herausgesucht. Dieser Wert kommt im Akkumulator zurück.

Um jetzt tatsächlich auf eine Bank umzuschalten, muß der zurückgegebene Akkumulatorwert in die Speicherzelle 65280 ($FF00) geschrieben werden. Im Beispiel wird der MMU-Wert von Bank 1 in die Speicherzelle 250 geschrieben.

```
ERG    =    250
GETCFG =    $FF6B
       LDX  #1                  ;X-REGISTER=BANKNUMMER=1
       JSR  GETCFG              ;BITKOMBINATION HOLEN
       STA  ERG                 ;DAS ERGEBNIS SPEICHERN
       RTS                      ;DAS PROGRAMM VERLASSEN
```

$FF6E – JSRFAR – Unterprogramm in einer anderen Bank aufrufen

Ist ein Programm auf mehrere Banks verteilt, können die einzelnen Teile durch Sprünge nur sehr schwer verbunden werden. Eine spezielle Routine des Betriebssystems übernimmt diese Aufgabe. Beim Aufruf der Routine muß das X-Register die Banknummer des Unterprogramms enthalten. Startadresse und Registerinhalte für den Unterprogrammstart werden in folgenden Speicherzellen mitgeteilt:

3: LO-Byte der Startadresse
4: HI-Byte der Startadresse
5: Statusregister
6: Akkumulator
7: X-Register
8: Y-Register

Das aufgerufene Unterprogramm kann ganz normal mit einem RTS-Befehl enden.

$FF71 – JMPFAR – Sprung zu einem Programm in einer anderen Bank

Diese Routine hat denselben Effekt wie die vorangegangene JSRFAR-Routine. Zum Programm wird jedoch ein Sprung ausgeführt. Startadresse und Registerinhalte werden in denselben Speicherzellen übergeben.

$FF74 – INDFET – LDA (...),Y aus einer beliebigen Bank

Mit dieser Routine können Daten aus einer beliebigen anderen Bank gelesen werden, ohne daß das Programm selbst auf diese Bank umschalten muß. In der Routine INDFET erfolgt der eigentliche Lesevorgang mit nach-indizierte, indirekter Adressierung (LDA (...),Y).

Zur Bestimmung der Speicherzelle muß die Leseadresse in zwei aufeinanderfolgenden Speicherzellen der Zero-Page, den ersten 256 Bytes, abgelegt sein. Die Adresse dieser Zero-Page-Speicherzellen wird im Akkumulator übergeben. Das X-Register enthält die Banknummer. Natürlich kann das Y-Register zusätzlich als Index verwendet werden. Das folgende Beispielprogramm liest ein Byte aus Bank 1 und gibt es in der Speicherzelle 250 zurück. Als Zeiger werden die Speicherzellen 251 und 252 benutzt.

```
ERG     =    250
ZEIGER  =    251
ADRESSE =    1024
INDFET  =    $FF74
        LDA  #<ADRESSE
        STA  ZEIGER
        LDA  #>ADRESSE
        STA  ZEIGER+1
        LDA  #ZEIGER
        LDX  #1
        LDY  #0
        JSR  INDFET
        STA  ERG
        RTS
```

$FF77 – INDSTA – STA (...),Y in eine beliebige Bank

Genau wie die vorherige Routine erlaubt auch INDSTA den Zugriff auf eine beliebige Speicherbank. Diesmal wird die Zero-Page-Adresse des Zeigers in der Speicherzelle 697 ($02B9) mitgeteilt. Das X-Register enthält wieder die Banknummer. Das zu schreibende Datenbyte wird im Akkumulator übergeben. Als Beispiel wird der Wert 111 in die Speicherzelle 1024 von Bank 1 geschrieben.

```
ZEIGER    =     251
ADRESSE   =     1024
STAZEIGER =     697
INDSTA    =     $FF77
          LDA   #<ADRESSE
          STA   ZEIGER
          LDA   #>ADRESSE
          STA   ZEIGER+1
          LDA   #ZEIGER
          STA   STAZEIGER
          LDX   #1
          LDY   #0
          LDA   #111
          JMP   INDSTA
```

$FF7A – INDCMP – CMP (...),Y mit einer beliebigen Bank

Diese Routine vergleicht den Akkumulatorinhalt mit dem Inhalt einer beliebigen Speicherzelle in einer beliebigen Bank. Die Adresse des Zero-Page-Zeigers wird in der Speicherzelle 712 ($02CD) festgesetzt.

Das X-Register enthält die gewünschte Banknummer. Nach Aufruf der Routine kann das Ergebnis des Vergleichs an Hand der Flags wie gewohnt überprüft werden. Das Beispiel prüft, ob sich in der Speicherzelle 1024 der Wert 111 befindet.

```
ZEIGER    =     250
ADRESSE   =     1024
CMPZEIGER =     712
INDCMP    =     $FF7A
BSOUT     =     $FFD2
          LDA   #<ADRESSE
          STA   ZEIGER
          LDA   #>ADRESSE
          STA   ZEIGER+1
          LDA   #ZEIGER
          STA   CMPZEIGER
          LDX   #1
          LDY   #0
          LDA   #111
```

```
            JSR   INDCMP
            BNE   ENDE
            LDA   #"="
            JSR   BSOUT
END         RTS
```

$FF7D – PRIMM – nachfolgenden Text ausgeben

Dies ist eine sehr interessante Routine. Zur Ausgabe fester Meldungen müßten extra Schleifen programmiert werden. Hier folgt im Objektcode der Text direkt dem Routinenaufruf und kann beliebig lang sein. Als Endmarkierung dient ein Nullbyte (0). Während der Ausgabe wird der Programmzeiger über den Text hinweggesetzt. Im Programm können dem Text also direkt weitere Befehle folgen. Das Beispiel gibt einen Text fünfmal aus.

```
PRIMM      =     $FF7D
           LDX   #5                      ;DEN SCHLEIFENZAEHLER STARTEN
SCHLEIFE   JSR   PRIMM                   ;DEN FOLGENDEN TEXT AUSGEBEN
           .TEXT "SYBEX-VERLAG",13,0     ;DER AUSGABETEXT
           DEX                           ;DEN SCHLEIFENZAEHLER VERMINDERN
           BNE   SCHLEIFE                ;NULL (ENDE) ?, NEIN =>
           RTS                           ;DAS PROGRAMM VERLASSEN
```

Die Routine kann nur für eine feststehende Meldung benutzt werden. Variable Texte, wie z.B. vom Programmbenutzer eingegebene Namen, können damit nicht ausgedruckt werden.

$FF81 – JPCINT – die Video-Chips und den Editor initialisieren

Diese Routine ruft die Editor-Funktion CINT auf.

$FF84 – IOINIT – I/O-Bausteine initialisieren

Diese Routine versetzt alle Ein-/Ausgabebausteine in eine Grundstellung, wie sie auch nach dem Einschalten des Computers besteht. In Programmen können Sie einen fest definierten Zustand der Bausteine herstellen.

$FF87 – RAMTAS – Initialisierung von RAM, Kassette und RS232

Mit dieser Routine werden alle Ein-/Ausgabepuffer, wie z.B. Kassettenpuffer oder RS232-Datenpuffer gelöscht. Zusätzlich wird wieder der maximale RAM-Speicher als verfügbar markiert (siehe MEMTOP, MEMBOT).

$FF8A – RESTOR – I/O-Vektoren auf Standardwert setzen

Wurden in einem Programm die I/O-Vektoren verändert, werden mit dieser Routine die Normalwerte wiederhergestellt.

$FF8D – VECTOR – I/O-Vektoren mit neuen Werten belegen

Mit dieser Routine werden alle 16 Vektoren auf einmal geändert. Die neuen Adressen sind in einer Tabelle abgelegt, deren Startadresse im X- (LO-Byte) und Y-Register (HI-Byte) übergeben wird. In der Tabelle stehen die Vektoren, aufgeteilt in LO- und HI-Byte, hintereinander.

$FF90 – SETMSG – Flag für Betriebssystemmeldung setzen

Außer dem BASIC-Interpreter kann auch das Betriebssystem selbst Fehlermeldungen ausgeben. Diese werden aus dem Text "I/O ERROR #" und einer nachfolgenden Zahl gebildet, die die Art des Fehlers angibt. Der Inhalt einer Speicherzelle bestimmt, ob eine Meldung ausgegeben oder unterdrückt wird. Der neue Wert wird im Akkumulator übergeben.

Die Ausgabe der Meldung wird beim Wert 64 (Bit 6 gesetzt) verhindert und beim Wert 0 (Bit 6 gelöscht) erlaubt. Außerdem kann die Ausgabe der Meldungen bei Lade- und Speichervorgängen ("SEARCHING FOR",...) mit dem Wert 128 (Bit 7 gesetzt, Bit 6 gelöscht) bzw. 192 (Bit 7 und Bit 6 gesetzt) unterbunden werden.

Den Nummern der Fehlermeldungen entsprechen folgende BASIC-Fehlermeldungen:

1: too many files
2: file open
3: file not open
4: file not found
5: device not present
6: not input file
7: not output file
8: missing filename
9: illegal device number
A: out of memory

Egal, ob eine Ausgabe erfolgte oder nicht, nach einem Fehler ist das Übertrags-Flag (C-Flag) gesetzt, und der Akkumulator enthält die Fehlernummer. Der Wert 16 entspricht dabei der Fehlernummer "A".

$FF93 – LSTNSA – Sekundäradresse nach LISTEN ausgeben

Mit dieser Routine wird der IEC-Bus direkt angesprochen. Dem angesprochenen Peripheriegerät wird nach einem LISTEN-Kommando mit dieser Routine die Sekundäradresse mitgeteilt. Näheres über die Bedienung des IEC-Busses erfahren Sie in einem eigenen Abschnitt.

$FF96 – TALKSA – Sekundäradresse nach TALK ausgeben

Auch mit dieser Routine wird der IEC-Bus direkt angesprochen. Es wird die Sekundäradresse nach einem TALK-Kommando übermittelt. Näheres erfahren Sie in einem eigenen Kapitel.

$FF99 – MEMTOP – Speicherobergrenze lesen bzw. schreiben

Über diese Routine bestimmt das Betriebssystem die Obergrenze des verfügbaren Speichers in Bank 0. Bei gelöschtem Übertrags-Flag wird ein neuer Wert geschrieben und bei gesetztem Flag gelesen. Das X-Register enthält jeweils das LO-Byte und das Y-Register das HI-Byte. Der BASIC-Interpreter bestimmt z.B. damit die Obergrenze seines Programmspeichers.

$FF9C – MEMBOT – Speicheruntergrenze lesen bzw. schreiben

Diese Routine bestimmt entsprechend die Untergrenze des verfügbaren Speichers in Bank 0. Bei gelöschtem Flag wird der Wert geschrieben und bei gesetztem Flag gelesen. Das X-Register übergibt das LO-Byte und das Y-Register das HI-Byte.

$FFA5 – IECIN – ein Zeichen vom IEC-Bus lesen

Diese Routine liest ein Zeichen vom IEC-Bus. Zuvor muß jedoch mit TALK ein sendendes Gerät bestimmt worden sein. Detaillierte Informationen zum IEC-Bus erhalten Sie in einem eigenen Abschnitt.

$FFA8 – IECOUT – ein Zeichen auf dem IEC-Bus ausgeben

Diese Routine gibt ein Zeichen auf dem IEC-Bus aus. Zuvor muß jedoch mit LISTEN ein Empfänger für das Byte bestimmt worden sein. Näheres zum Ablauf einer Datenübertragung über den IEC-Bus erfahren Sie in einem eigenen Abschnitt.

$FFAB – UNTALK – UNTALK-Kommando auf dem IEC-Bus senden

Mit dieser Routine wird die Datenübertragung von einem Peripheriegerät zum Computer beendet. Genaueres hierzu erfahren Sie in einem eigenen Kapitel.

$FFAE – UNLSTN – UNLISTEN-Kommando auf dem IEC-Bus senden

Diese Routine beendet eine Datenübertragung vom Computer zu einem Peripheriegerät des IEC-Busses.

$FFB1 – LISTEN – LISTEN-Kommando auf dem IEC-Bus senden

Durch diese Routine wird ein Peripheriegerät am IEC-Bus zum Empfang von Daten vorbereitet.

$FFB4 – TALK – TALK-Kommando auf dem IEC-Bus senden

Diese Routine leitet die Datenübertragung von einem Peripheriegerät des IEC-Busses zum Computer ein.

$FFB7 – READST – aktuelles Statusbyte lesen

Der Status einer gerade laufenden Ein-/Ausgabeoperation kann mit dieser Routine gelesen werden. Jedem Bit des Statuswortes kommt folgende Bedeutung zu:

Bit 0: Zeitüberschreitung (Timeout) beim Schreiben
1: Zeitüberschreitung (Timeout) beim Lesen
2: zu kurzer Datenblock auf der Kassette
3: zu langer Datenblock auf der Kassette
4: schwerwiegender Fehler
5: Prüfsummenfehler
6: Dateiende erreicht (keine eigentliche Fehlermeldung)
7: Gerät nicht angeschlossen oder Bandende bei Kassette

Bei der Abfrage des IEC-Status wird man in Assemblerprogrammen nicht diese Routine verwenden, sondern direkt die entsprechende Speicherzelle (144 = $90) in der Zero-Page lesen. Die Bedeutung der Bits ist identisch. Wurde der letzte Datenverkehr über die RS232-Schnittstelle abgewickelt, wird der RS232-Status zurückgegeben. Die einzelnen Bits haben dann folgende Bedeutung:

Bit 0: Paritätsfehler
1: Rahmenfehler
2: Empfängerpuffer voll
3: -- unbenutzt --
4: CTS-Signal (clear to send) fehlt
5: -- unbenutzt --
6: DSR-Signal (Data set ready) fehlt
7: Break-Signal fehlt

Übrigens entspricht der Wert dieses Statusbytes dem der BASIC-Systemvariablen ST.

$FFBA – SETLFS – LA, FA, SA einer Datei bestimmen

Zum Öffnen einer Datei wird eine logische Dateinummer (Akkumulator), eine Geräteadresse (X-Register) und eine Sekundäradresse (Y-Register) benötigt. Mit dieser Routine werden dem Betriebssystem die neuen Parameter mitgeteilt. Wie die einzelnen Schritte zum Eröffnen einer Datei lauten, wird ausführlich in einem eigenen Abschnitt beschrieben.

$FFBD – SETNAM – Dateiname übergeben

Mit dieser Routine wird dem Betriebssystem der Dateiname und dessen Länge mitgeteilt. Der Name muß im Speicher abgelegt sein. Das X- (LO-Byte) und Y-Register (HI-Byte) enthalten die Startadresse. Der Akkumulator übergibt die Länge des Dateinamens. Wird an dieser Stelle eine Null übergeben, entfällt z.B. bei einem nachfolgenden Öffnen einer Datei der Dateiname. Neben diesen Parametern muß mit der Routine SETBNK die Speicherbank bestimmt werden, in der der Name abgelegt ist. Die weiteren Dateiparameter werden dem Betriebssystem mit der Routine SETLFS mitgeteilt.

$FFC0 – OPEN – eine Datei öffnen

Diese Routine öffnet eine neue Datei. Die verschiedenen Parameter, wie Geräteadresse oder Dateiname, müssen zuvor mit den Routinen SETLFS und SETNAM definiert werden. Die Routine trägt die Dateiparameter in eine Liste ein und öffnet die Datei auf dem angegebenen Peripheriegerät. Ist am Routinen-Ende das Übertrags-Flag (C-Flag) gesetzt, trat beim Öffnen der Datei ein Fehler auf. Beispiele zum Arbeiten mit Dateien in Assemblersprache finden Sie in einem eigenen Abschnitt dieses Kapitels.

$FFC3 – CLOSE – eine Datei schließen

Eine zuvor geöffnete Datei wird mit dieser Routine wieder geschlossen. Der Akkumulator muß die logische Dateinummer enthalten. Ein gesetztes Übertrags-Flag (C-Flag) nach Abschluß der Routine weist auf einen Fehler hin.

$FFC6 – CHKIN – einen Eingabekanal öffnen

Nachdem eine Datei mit der Routine OPEN geöffnet wurde, wird mit CHKIN die Eingabe auf diese Datei umgelenkt. Beim Aufruf dieser Routine muß das X-Register die logische Dateinummer enthalten. Die Anwendung dieser Routine wird in einem eigenen Abschnitt beschrieben.

$FFC9 – CKOUT – einen Ausgabekanal öffnen

Mit dieser Routine wird die Ausgabe auf eine zuvor geöffnete Datei umgelenkt. Das X-Register muß auch hier die logische Dateinummer enthalten. Beispiele zur Anwendung der Routine sind in einem eigenen Abschnitt abgedruckt.

$FFCC – CLRCH – Ein- und Ausgabekanal schließen

Diese Routine ist das Gegenstück zu CHKIN und CKOUT. Die geöffneten Ein- und Ausgabekanäle werden geschlossen. Die Eingabe wird wieder auf die Tastatur gelegt und die Ausgabe auf den Bildschirm umgelenkt.

$FFCF – BASIN – ein Zeichen vom aktiven Kanal einlesen

Die Routine liest ein Zeichen vom aktuellen Eingabeort. Dies ist im Normalfall die Tastatur. Der Eingabekanal kann mit der Routine CHKIN jedoch neu bestimmt werden. Ist das Eingabegerät die Tastatur, werden zunächst Zeichen von der Tastatur gelesen und auf dem Bildschirm dargestellt. Nach Drücken der Taste RETURN wird der Inhalt der Zeile, in der sich der Cursor gerade befindet, Zeichen für Zeichen gelesen, beim ersten Aufruf der Routine das erste Zeichen, beim zweiten Aufruf das zweite usw. Schlußzeichen ist ein Wagenrücklauf (13). Leerstellen werden vom Zeilenende automatisch entfernt.

Erfolgt die Eingabe vom Bildschirm, werden die Zeichen sofort, ab der Cursorposition, vom Bildschirm gelesen. Schlußzeichen ist wieder ein Wagenrücklauf. Leerstellen werden hier nicht entfernt. Es werden also immer bis

zum Zeilenende Zeichen gelesen. Als Beispiel wird eine Zeile mit der Tastatur als Eingabegerät in einen Puffer gelesen. Dieser wird dann ausgegeben.

```
CRSRON     =    $CD6F
BASIN      =    $FFCF
BSOUT      =    $FFD2
           JSR  CRSRON          ;DEN CURSOR EINSCHALTEN
           LDX  #0              ;INDEX=0
SCHLEIFE   JSR  BASIN           ;ZEICHEN VON DER TASTATUR LESEN
    STA    PUFFER,X             ;DAS ZEICHEN MERKEN
    CMP    #13                  ;MIT WAGENRUECKLAUF VERGLEICHEN
    BEQ    AUSGABE              ;GLEICH ?, JA =>
    INX                         ;DEN INDEX ERHOEHEN
    JMP    SCHLEIFE             ;ZUM SCHLEIFENANFANG SPRINGEN
;
AUSGABE    LDX  #0              ;INDEX=0
SCHLEIFE2  LDA  PUFFER,X        ;ZEICHEN AUS DEM PUFFER LESEN
           JSR  BSOUT           ;DAS ZEICHEN AUSGEBEN
           CMP  #13             ;MIT WAGENRUECKLAUF VERGLEICHEN
           BEQ  ENDE            ;GLEICH ?, JA =>
           INX                  ;DEN INDEX ERHOEHEN
           JMP  SCHLEIFE2       ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                  ;DAS PROGRAMM VERLASSEN
;
PUFFER     =    *               ;PUFFER ANS PROGRAMMENDE LEGEN
```

$FFD2 – BSOUT – ein Zeichen auf dem aktiven Kanal ausgeben

Mit dieser Routine wird ein Zeichen auf dem aktuellen Ausgabekanal ausgegeben. Im Normalfall ist dies der Bildschirm. Mit der Routine CKOUT kann jedoch jederzeit ein anderes Gerät zur Ausgabe bestimmt werden. Ein Anwendungsbeispiel sehen Sie z.B. bei der Routine BASIN.

$FFD5 – LOADSP – LOAD oder VERIFY durchführen

Vor dem Aufruf dieser Routine müssen mit den Routinen SETLFS und SETNAM die Dateiparameter gesetzt werden. Beim Aufruf selbst bestimmt der Akkumulator, ob die Datei geladen (Akkumulator=0) oder mit dem Speicher verglichen (Akkumulator<>0) werden soll. Ist die verwendete Sekundäradresse ungleich null, wird ab der in der Datei eingetragenen Adresse geladen bzw. verglichen.

Bei einer Sekundäradresse gleich null wird die beim Aufruf vorgegebene Adresse verwendet. Das X-Register enthält das LO-Byte und das Y-Register das HI-Byte dieser Adresse. Als Beispiel wird ein BASIC-Programm an die dafür übliche Adresse $1C01 geladen.

```
SETBNK  =     $FF68
GETCFG  =     $FF6B
SETLFS  =     $FFBA
SETNAM  =     $FFBD
LOADSP  =     $FFD5
START   =     $1C01
        LDX   #0 ;BANKNUMMER=0
        JSR   GETCFG            ;MMU-KOMBINATION HOLEN
        JSR   SETBNK            ;DIE BANK AUSWAEHLEN
        LDA   #1                ;LOGISCHE DATEINR. (HIER EGAL)
        LDX   #8                ;X-REGISTER=GERAETEADR.=8
        LDY   #0                ;Y-REGISTER-SEKUNDAERADR.=0
        JSR   SETLFS            ;DIE PARAMETER UEBERGEBEN
        LDA   #8                ;AKKUMULATOR=LAENGE=7
        LDX   #<NAME            ;LO-BYTE DER STARTADR. DES NAMENS
        LDY   #>NAME            ;HI-BYTE DER STARTADR. DES NAMENS
        JSR   SETNAM            ;DIE PARAMETER UEBERGEBEN
        LDA   #0                ;AKKUMULATOR=LOAD/VERIFY=0
        LDX   #<START           ;LO-BYTE DES PROGRAMMSTARTS
        LDY   #>START           ;HI-BYTE DES PROGRAMMSTARTS
        JMP   LOADSP            ;PROGRAMM LADEN +PROGRAMMENDE
NAME            .TEXT "PROGRAMM"  ;NAME DES PROGRAMMS
```

Auf diesem Weg sollten nur Maschinenprogramme geladen werden, da bei BASIC-Programmen Mitteilung an den BASIC-Interpreter fehlen, wie das neue Ende BASIC-Textes.

$FFD8 – SAVESP – SAVE durchführen

Diese Routine speichert einen angegebenen Teil des Speichers auf ein Peripheriegerät. Mit den Routinen SETLFS und SETNAM müssen zuvor die Dateiparameter definiert werden. Die Startadresse des zu sichernden Speicherbereichs wird in der Zero-Page in zwei aufeinanderfolgenden Speicherzellen abgelegt. Die Adresse dieser Speicherzellen wird im Akkumulator übergeben.

Zusätzlich enthält das X-Register das LO-Byte und das Y-Register das HI-Byte der Endadresse. Als Beispiel wird der Speicherbereich von $1300-$19FF in die Datei "SPEICHER" geschrieben.

```
ZEIGER  =     250
SETBNK  =     $FF68
GETCFG  =     $FF6B
SETLFS  =     $FFBA
SETNAM  =     $FFBD
SAVESP  =     $FFD8
START   =     $1300
END     =     $19FF
        LDX   #0                ;BANKNUMMER=0
        JSR   GETCFG            ;MMU-KOMBINATION HOLEN
        JSR   SETBNK            ;DIE BANK AUSWAEHLEN
```

```
LDA   #1              ;LOGISCHE DATEINR. (HIER EGAL)
LDX   #8              ;GERAETEADRESSE
LDY   #1              ;SEKUNDAERADRESSE (HIER EGAL)
JSR   SETLFS          ;PARAMETER UEBERGEBEN
LDA   #8              ;LAENGE DES NAMENS
LDX   #<NAME          ;LO-BYTE DES NAMENANFANGS
LDY   #>NAME          ;HI-BYTE DES NAMENANFANGS
JSR   SETNAM          ;PARAMETER SETZEN
LDA   #<START         ;LO-BYTE DER STARTADRESSE
STA   ZEIGER          ;SPEICHERN
LDA   #>START         ;HI-BYTE DER STARTADRESSE
STA   ZEIGER+1        ;SPEICHERN
LDA   #ZEIGER         ;ADRESSE DER STARTADRESSE
LDX   #<END           ;LO-BYTE DER ENDADRESSE
LDY   #>END           ;HI-BYTE DER ENDADRESSE
JMP   SAVESP          ;SPEICHERN +PROGRAMMENDE
```

$FFDB – SETTIM – die Systemuhr neu setzen

In der Interrupt-Routine wird außer der Tastaturabfrage auch die interne Systemuhr erhöht. Dabei handelt es sich um nichts anderes als einen drei Byte großen Zähler. Beim Einschalten des Computers beginnt die Zählung bei null. Ein Interrupt wird jede Sechzigstel-Sekunden ausgelöst. Ein Zählerwert von 60 entspricht demnach einer Sekunde, 3600 einer Minute und 216000 einer Stunde. Mit dieser Routine kann die Systemuhr auf einen neuen Wert gesetzt werden. Zur Berechnung der einzelnen Werte für die drei Bytes noch folgende Information: Das niederwertigste Byte (Y-Register) hat den Stellenwert 1, das mittlere Byte (X-Register) den Stellenwert 256 und das höchste Byte (Akkumulator) den Stellenwert 65536. Als Beispiel sollen zwei Stunden, 37 Minuten und 21 Sekunden eingestellt werden.

$$n = 21 * 60 + 37 * 3600 + 2 * 216000 = 566460$$

$$\begin{aligned} 566460/65536 &= 8.64... &&\Rightarrow 8 \\ 566460 - 8 * 65536 &= 42172/256 = 164.73... &&\Rightarrow 164 \\ 42172 - 164 * 256 &= 188 &&\Rightarrow 188 \end{aligned}$$

```
SETTIM  =     $FFDB
        LDY   #188              ;UNTERES BYTE
        LDX   #164              ;MITTLERES BYTE
        LDA   #8                ;OBERES BYTE
        JMP   SETTIM            ;SYSTEMUHR SETZEN UND PROGRAMMENDE
```

$FFDE – RDTIM – die Systemuhr lesen

Diese Routine ist das Gegenstück zur gerade beschriebenen SETTIM-Routine. Es wird die Systemuhr ausgelesen. Das Ergebnis wird im Akkumulator (obe-

res Byte), im X-Register (mittleres Byte) und im Y-Register (unteres Byte) zurückgeliefert. Die Umrechnungsmethode lesen Sie bitte bei der vorherigen Routine nach.

$FFE1 – STOP – die STOP-Taste prüfen

Diese Routine prüft, ob die Stop-Taste gedrückt ist. Das Ergebnis wird im Null-Flag (Z-Flag) zurückgegeben. Ist das Flag gesetzt, wurde die Taste gedrückt.

Zu beachten ist, daß die Routine, im Fall einer gedrückten STOP-Taste den Ein- und Ausgabekanal auf die Tastatur bzw. den Bildschirm legt. Das Beispielprogramm verharrt bis zum Drücken der STOP-Taste in einer Endlosschleife.

```
STOP        =     $FFE1
SCHLEIFE    JSR   STOP         ;DIE STOP-TASTE PRUEFEN
            BNE   SCHLEIFE     ;NULL (GEDRUECKT) ?, NEIN =>
            RTS                ;DAS PROGRAMM VERLASSEN
```

$FFE4 – GETIN – ein Zeichen vom aktuellen Kanal lesen

Diese Routine hat die gleiche Funktion wie BASIN. Bei der Eingabe von der Tastatur werden jedoch tatsächlich die eingegebenen Zeichen gelesen. Ist beim Routinenaufruf keine Taste gedrückt bzw. kein Tastencode im Tastaturpuffer gespeichert, wird der Wert Null zurückgesandt.

Auch das Arbeiten mit der RS232-Schnittstelle hat sich geändert. Alle ankommenden Zeichen werden immer in einem Puffer zwischengespeichert. GETIN wartet bei einem leeren Puffer nicht auf ankommende Zeichen, sondern gibt den Wert Null zurück und signalisiert im Statusbyte, daß keine Daten empfangen wurden.

$FFE7 – CLALL – Filetabelle löschen, Ein-/Ausgabe zurücksetzen

Diese Routine schließt alle geöffneten Dateien und schaltet die Eingabe auf die Tastatur und die Ausgabe auf den Bildschirm. Zu beachten ist, daß es sich hierbei um kein richtiges Schließen der Dateien handelt.

Es werden nur die internen Tabellen im Betriebssystem gelöscht und nicht die Datei beim Peripheriegerät geschlossen. Für diesen Zweck müssen Sie immer die Routine CLOSE verwenden.

$FFEA – CLOCK – Systemuhr aktualisieren

Diese Routine wird bei jedem Interrupt aufgerufen, um die Systemuhr zu erhöhen. Sollten Sie eine modifizierte Interrupt-Routine verwenden, muß diese Routine aufgerufen werden, damit die Systemuhr korrekt weiterläuft.

$FFED – JSCORG – aktuelle Bildschirmgröße lesen

Hiermit wird die Routine SCRORG des Editors aufgerufen.

$FFF0 – JPPLOT – die Cursorposition lesen bzw. schreiben

Mit dieser Adresse wird die Routine PLOT des Editors aufgerufen.

$FFF3 – IOBAS – Basis-Adresse der I/O-Bausteine lesen

Diese Routine gibt im X- (LO-Byte) und Y-Register (HI-Byte) die Basisadresse der Ein-/Ausgabebausteine zurück. Im C128 ist dies die Adresse $D000. In Programmen wird man diese Routine nie benötigen.

Die Vektoren von BASIC und Betriebssystem

Viele Routinen werden nicht direkt angesprochen, sondern laufen zuvor noch über einen sogenannten Vektor. Im ROM wird einfach ein indirekter Sprung der Form JMP (...) ausgeführt. Die Sprungadresse selbst ist nicht im ROM, sondern im RAM abgelegt und kann folglich verändert werden.

Durch solche Eingriffe können Zusatzprogramme, wie Befehlserweiterungen oder Interfaceprogramme, mit den normalen Routinen des Computers verbunden werden. Es folgt jetzt eine Kurzbeschreibung der einzelnen Vektoren.

BASIC-Vektoren:

$0300 – IERROR – Fehler-Routine

Dieser Vektor wird beim Auftreten eines BASIC-Fehlers verwendet. Das X-Register enthält die Fehlernummer. Beim Wert 128 ($80) handelt es sich nicht um einen Fehler, sondern um die Beendigung des laufenden BASIC-Programms.

$0302 – IMAIN – BASIC-Warmstart

Dieser Vektor wird nach der Ausgabe von ".READY" benutzt. Direkt folgend beginnt die Befehlseingabe. EDASS z.B. benutzt diesen Vektor für die eigene Befehlserweiterung.

$0304 – ICRNCH – Token-Umwandlung

Jedes BASIC-Befehlswort wird in einen Zahlencode, ein sogenannte Token, umgewandelt. Nach Aufruf der entsprechenden Routine wird über diesen Vektor gesprungen.

$0306 – IQPLOP – BASIC-Text listen

Innerhalb der LIST-Routine des BASIC-Interpreters wird zur Umwandlung und Ausgabe eines Tokens über diesen Vektor gesprungen.

$0308 – IGONE – BASIC-Befehl ausführen

Innerhalb der eigentlichen Interpreter-Routine, in der die einzelnen Befehle aufgerufen werden, wird über diesen Vektor gesprungen.

$030A – IEVAL – Ausdruck auswerten

Im BASIC-ROM befindet sich eine Routine, die einen beliebigen mathematischen Ausdruck auswertet. Innerhalb dieser wird der Vektor EVAL verwendet.

Betriebssystem-Vektoren:

$0314 – IIRQ – Interrupt-Vektor

Nach Auftreten eines Interrupts springt der Prozessor in die Interrupt-Routine. Innerhalb dieser wird mit dem IRQ-Vektor zur eigentlichen Bearbeitungsroutine verzweigt.

$0316 – IBRK – Break-Vektor

Bei einem BRK-Befehl wird ebenfalls ein Interrupt ausgelöst. Der Prozessor verzweigt in die Interrupt-Routine und innerhalb dieser über diesen Vektor. Im Normalfall zeigt der Vektor auf den Monitoreinsprung ($B003).

$0318 – INMI – NMI-Vektor

Bei Auftreten eines NMI-Interrupts wird zuerst in die entsprechende ROM-Routine verzweigt und über diesen Vektor zur eigentlichen Bearbeitung. Die RESTORE-Taste ist mit der NMI-Leitung verbunden. In der ROM-Routine vor der Vektorbenutzung wird noch nicht geprüft, ob die RUN/STOP-Taste gedrückt ist.

$031A – IOPEN – Vektor in der OPEN-Routine

Über diesen Vektor wird die OPEN-Routine angesprochen. Der Vektor-Aufruf liegt direkt in der Sprungleiste.

$031C – ICLOSE – Vektor in der CLOSE-Routine

Über diesen Vektor wird in der Sprungleiste die Routine CLOSE aufgerufen.

$031E – ICHKIN – Vektor zum Öffnen des Eingabekanals

Genau wie die zwei vorangegangenen Vektoren wird auch mit diesem direkt in der Sprungleiste die Routine CHKIN aufgerufen.

$0320 – ICKOUT – Vektor zum Öffnen des Ausgabekanals

Dieser Vektor ruft direkt in der Sprungleiste die Routine CKOUT auf.

$0322 – ICLRCH – Vektor, um Ein-/Ausgabekanal zurückzusetzen

Über diesen Vektor wird die Routine CLRCH aufgerufen. Die Verwendung erfolgt direkt in der Sprungleiste.

$0324 – IBASIN – Vektor zum Lesen eines Zeichens

Dieser Vektor dient zum Aufruf der Routine BASIN.

$0326 – IBSOUT – Vektor zur Ausgabe eines Zeichens

In der Sprungleiste wird über diesen Vektor die Routine BSOUT zur Ausgabe eines Zeichens aufgerufen.

$0328 – ISTOP – Vektor bei der Abfrage der STOP-Taste

Bei der Abfrage der STOP-Taste mit der Routine STOP wird über diesen Vektor die Routine aufgerufen.

$032A – IGETIN – Vektor zum Einlesen eines Zeichens

Die zweite Routine zum Lesen eines Zeichens, GETIN, wird über diesen Vektor aufgerufen.

$032C – ICLALL – Vektor zum Schließen aller Dateien

Beim Schließen aller Dateien über die Routine CLALL wird diese über den Vektor ICLALL aufgerufen.

$032E – EXMON – Einsprung in den Monitor

Der eingebaute Maschinensprache-Monitor kann über diesen Vektor aufgerufen werden.

$0330 – ILOAD – Vektor in der LOAD-Routine

Dieser Vektor wird in der LOAD-Routine benutzt. Zwischen dem Aufruf der Sprungleiste und der Verwendung des Vektors wird die vorgegebene Startadresse in den Speicherzellen 195 ($C3) und 196 ($C4) gespeichert.

$0332 – ISAVE – Vektor in der SAVE-Routine

Dieser Vektor wird innerhalb der SAVE-Routine aufgerufen. Zwischen dem Aufruf über die Sprungleiste und der Verwendung des Vektors wird die End-

adresse in den Speicherzellen 174 ($AE) und 175 ($AF) gespeichert. Die Startadresse wird gelesen und in den Speicherzellen 193 ($C1) und 194 ($C2) abgelegt. Zwei Beispiele für das Verändern der Vektoren finden Sie in den Programmen, die auf der Originaldiskette mitgeliefert werden. Das eine ist ein Interfaceprogramm, das am User-Port eine Centronics-Schnittstelle erzeugt, und das zweite ist ein Programm, das die Umlaute des Commodore ASCII-Codes für einen Standard-ASCII-Drucker aufbereitet, damit diese dort korrekt erscheinen.

Arbeiten mit Dateien in Assemblerprogrammen

In diesem Abschnitt wird das Zusammenwirken der verschiedenen oben beschriebenen Routinen für die Dateiverwaltung erklärt. Das Betriebssystem benötigt zum Verwalten aller geöffneten Dateien eine logische Dateinummer, eine Geräteadresse und eine Sekundäradresse. Unter der logischen Dateinummer wird die Datei immer angesprochen.

Es dürfen deshalb auch keine Dateien mit gleicher logischer Nummer geöffnet werden. Um die Datei auf dem Peripheriegerät, speziell der Diskette, zu öffnen, muß ein Dateiname eingegeben werden. Diese zwei Gruppen von Parametern werden über die Routinen SETLFS und SETNAM dem Betriebssystem mitgeteilt. Das eigentliche Öffnen der Datei, d.h. das Eintragen der Parameter in die interne Liste und die Übermittlung an das Peripheriegerät, erfolgt mit der Routine OPEN. Zu beachten ist, daß in diesen Schritten für das Betriebssystem nicht festgelegt ist, ob über die Datei gelesen oder geschrieben wird. Hier ein paar Beispiele für das Öffnen einer Datei:

```
LDA    #2                    ;LOGISCHE DATEINUMMER=2
LDX    #8                    ;GERAETEADRESSE=8
LDY    #4                    ;SEKUNDAERADRESSE=4
JSR    SETLFS                ;PARAMETER SETZEN
LDA    #11                   ;LAENGE DES NAMENS
LDX    #<NAME                ;LO-BYTE DER ADRESSE DES NAMENS
LDY    #>NAME                ;HI-BYTE DER ADRESSE DES NAMENS
JSR    SETNAM                ;NAMEN UEBERGEBEN
JSR    OPEN                  ;DIE DATEI OEFFNEN
...
NAME   .TEXT"0:DATEI,S,R"    ;DATEINAME

LDA    #1                    ;LOGISCHE DATEINUMMER=1
LDX    #4                    ;GERAETEADRESSE=4
LDY    #0                    ;SEKUNDAERADRESSE=0
JSR    SETLFS                ;PARAMETER UEBERGEBEN
LDA    #0                    ;LAENGE DES NAMENS=0
JSR    SETNAM                ;NAMEN UEBERGEBEN
JSR    OPEN                  ;DATEI OEFFNEN
...
```

Im letzten Beispiel wurde eine Datei zum Drucker geöffnet. In diesem Fall wird üblicherweise kein Name angegeben. Dies wird dem Betriebssystem durch den Wert Null als Namenslänge mitgeteilt. Eine Adresse auf den Anfang des Namens ist dann natürlich auch nicht erforderlich. Noch ein Hinweis zur Verwendung von Dateinamen: Dem Betriebssystem muß mitgeteilt werden, in welcher Speicherbank der Name abgelegt ist. Durch folgende zwei Routinenaufrufe wird dies erreicht:

```
LDX  #1                    ;BANKNUMMER LADEN (HIER BANK 1)
JSR  GETCFG                ;MMU-KOMBINATION HOLEN
JSR  SETBNK                ;DIE BANK AUSWAEHLEN
...
```

Das Betriebssystem bezieht alle Eingaben aus dem Eingabekanal und leitet alle Ausgaben auf den Ausgabekanal. Normalerweise wird damit die Tastatur und der Bildschirm angesprochen. Es kann jedoch jede geöffnete Datei für die Ein- bzw. Ausgabe bestimmt werden. Dies ist auch der einzige Weg, Daten aus einer Datei zu lesen bzw. in eine Datei zu schreiben. Die Routine CHKIN bestimmt eine Datei als Eingabeort und CKOUT entsprechend eine Datei als Ausgabeort. Die logische Dateinummer der gewünschten Datei wird im X-Register übergeben. Die Ausgaberoutine BSOUT schreibt die Zeichen immer in den gerade aktiven Ausgabekanal. Die beiden Routinen BASIN und GETIN lesen die Daten vom aktiven Eingabekanal.

Die Routine CLRCH setzt sowohl den Ein- als auch den Ausgabekanal wieder auf die Tastatur bzw. den Bildschirm. Solange die Datei geöffnet ist, kann beliebig oft der Ein-/Ausgabekanal neu definiert werden. Auf diese Weise kann z.B. die Hälfte der Datei auf dem Bildschirm ausgegeben werden und die andere Hälfte auf einem Drucker. Soll z.B. die Ausgabe auf beide Geräte gleichzeitig erfolgen, müssen die Geräte nacheinander angesprochen werden, also zuerst den Ausgabekanal auf die Druckdatei legen und dann durch Schließen des Kanals auf den Bildschirm.

Am Ende der Datenübertragung wird die Datei wieder mit der Routine CLOSE geschlossen. Die logische Dateinummer wird hier im Akkumulator übergeben. Als erstes Beispiel wird das gesamte Alphabet in die Datei "ALPHABET" auf der Diskette geschrieben. Zur Verdeutlichung der einzelnen Schritte sehen Sie in Abb. 9.1 die grafische Darstellung des Programmablaufs.

```
SETBNK    =    $FF68
GETCFG    =    $FF6B
SETLFS    =    $FFBA
SETNAM    =    $FFBD
OPEN      =    $FFC0
CLOSE     =    $FFC3
CKOUT     =    $FFC9
CLRCH     =    $FFCC
BSOUT     =    $FFD2
```

```
CLRCH     =    $FFCC
BSOUT     =    $FFD2
          LDX  #0              ;BANKNUMMER=0
          JSR  GETCFG          ;MMU-KOMBINATION HOLEN
          JSR  SETBNK          ;DIE BANK AUSWAEHLEN
          LDA  #2              ;LOGISCHE DATEINUMMER=2
          LDX  #8              ;GERAETEADRESSE=8
          LDY  #2              ;SEKUNDAERADRESSE=2
          JSR  SETLFS          ;PARAMETER SETZEN
          LDA  #14             ;LAENGE DES NAMENS
          LDX  #<NAME          ;LO-BYTE DER ADRESSE DES NAMENS
          LDY  #>NAME          ;HI-BYTE DER ADRESSE DES NAMENS
          JSR  SETNAM          ;NAMEN UEBERGEBEN
          JSR  OPEN            ;DIE DATEI OEFFNEN
;
          LDX  #2              ;DATEINUMMER=2
          JSR  CKOUT           ;DEN AUSGABEKANAL BESTIMMEN
          LDX  #65             ;STARTWERT=ASCII-CODE VON "A"
SCHLEIFE  TXA                  ;DEN ASCII-CODE IN DEN AKKU.
          JSR  BSOUT           ;DAS ZEICHEN AUSGEBEN
          INX                  ;SCHLEIFENZAEHLER EROEÖHEN
          CPX  #91             ;MIT ENDBEDINGUNG VERGLEICHEN
          BNE  SCHLEIFE        ;ERREICHT ?,NEIN =>
          JSR  CLRCH           ;DEN AUSGABEKANAL SCHLIESSEN
;
          LDA  #2              ;DATEINUMMER=2
          JMP  CLOSE           ;DATEI SCHLIESSEN +PROGRAMMENDE
NAME      .TEXT"0:ALPHABET,S,W" ;DATEINAME
```

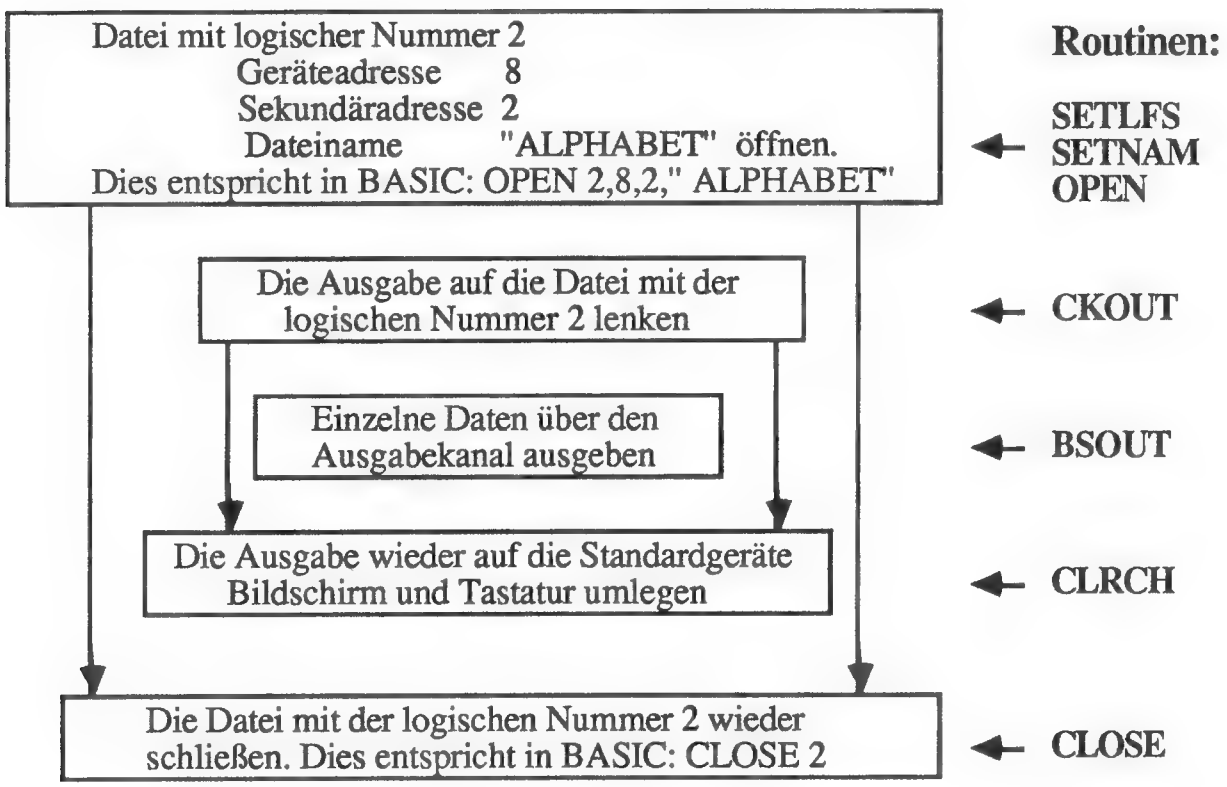


Abb. 9.1: Grafische Darstellung des Programmablaufes "ALPHABET"

Als zweites Beispiel wird die Datei "ALPHABET" wieder gelesen und auf dem Drucker ausgegeben.

```
SETBNK    =    $FF68
GETCFG    =    $FF6B
SETLFS    =    $FFBA
SETNAM    =    $FFBD
OPEN      =    $FFC0
CLOSE     =    $FFC3
CHKIN     =    $FFC6
CKOUT     =    $FFC9
CLRCH     =    $FFCC
BASIN     =    $FFCF
BSOUT     =    $FFD2
          LDA  #1              ;LOGISCHE DATEINUMMER=1
          LDX  #4              ;GERAETEADRESSE=4
          LDY  #0              ;SEKUNDAERADRESSE=0
          JSR  SETLFS          ;PARAMETER UEBERGEBEN
          LDA  #0              ;LAENGE DES NAMENS=0
          JSR  SETNAM          ;NAMEN UEBERGEBEN
          JSR  OPEN            ;DATEI OEFFNEN
;
          LDX  #0              ;BANKNUMMER=0
          JSR  GETCFG          ;MMU-KOMBINATION HOLEN
          JSR  SETBNK          ;DIE BANK AUSWAEHLEN
          LDA  #2              ;LOGISCHE DATEINUMMER=2
          LDX  #8              ;GERAETADRESSE=8
          LDY  #2              ;SEKUNDAERADRESSE=2
          JSR  SETLFS          ;PARAMETER SETZEN
          LDA  #14             ;LÄNGE DES NAMENS
          LDX  #<NAME          ;LO-BYTE DER ADRESSE DES NAMENS
          LDY  #>NAME          ;HI-BYTE DER ADRESSE DES NAMENS
          JSR  SETNAM          ;NAMEN ÜBERGEBEN
          JSR  OPEN            ;DIE DATEI ÖFFNEN
;
          LDX  #1              ;DATEINUMMER=1 (DRUCKER)
          JSR  CKOUT           ;DEN AUSGABEKANAL BESTIMMEN
          LDX  #2              ;DATEINUMMER=2 (DATEI)
          JSR  CHKIN           ;DEN EINGABEKANAL BESTIMMEN
SCHLEIFE  JSR  BASIN           ;EIN ZEICHEN LESEN
          JSR  BSOUT           ;DAS ZEICHEN AUSGEBEN
          CMP  #"Z"            ;WAR ES EIN "Z"
          BNE  SCHLEIFE        ;ERREICHT ?,NEIN =>
          LDA  #13             ;WAGENRUECKLAUF
          JSR  BSOUT           ;AUSGEBEN
          JSR  CLRCH           ;DEN AUSGABEKANAL SCHLIESSEN
;
          LDA  #1              ;DATEINUMMER=1
          JSR  CLOSE           ;DATEI SCHLIESSEN
          LDA  #2              ;DATEINUMMER=2
          JMP  CLOSE           ;DATEI SCHLIESSEN +PROGRAMMENDE
NAME      .TEXT"0:ALPHABET,S,R";DATEINAME
```

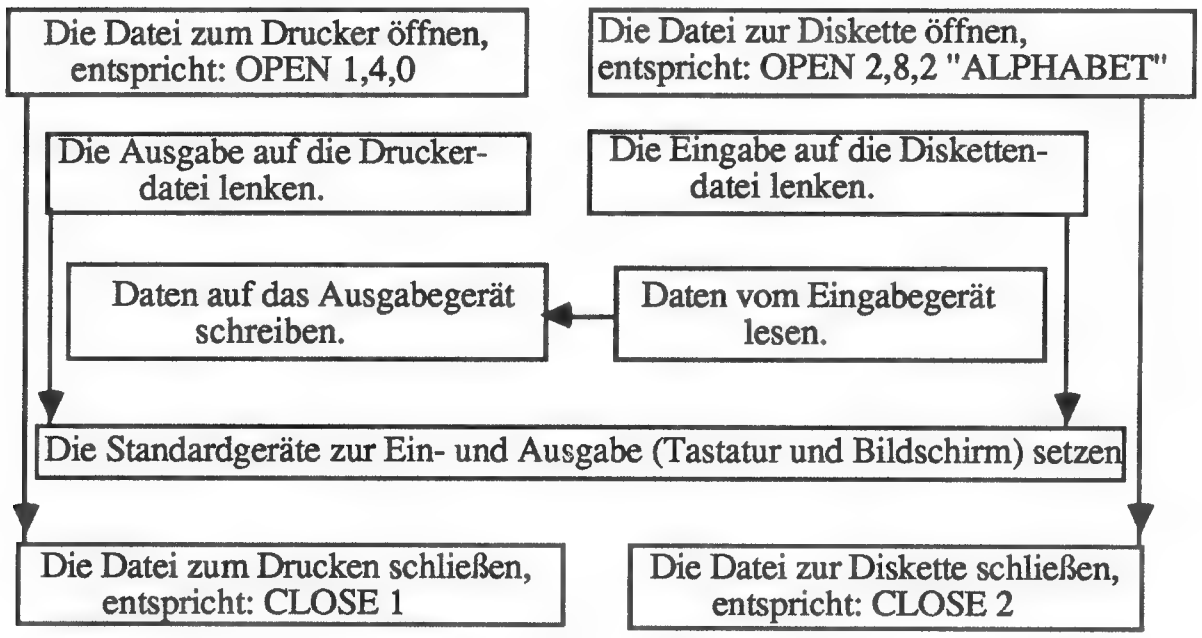


Abb. 9.2: Die Programmschritte in einer grafischen Darstellung

Zum Schluß noch ein Hinweis: Wird in einem Assemblerprogramm mit Dateien gearbeitet, müssen diese nicht unbedingt im Assemblerprogramm geöffnet worden sein. Dies kann z.B. ein BASIC-Programm übernehmen. Bei der Kanal-Bestimmung mit CHKIN oder CKOUT muß lediglich die im OPEN-Befehl verwendete logische Dateinummer verwendet werden.

Direkte Bedienung des IEC-Busses

Das Betriebssystem übernimmt die komplette Bedienung des IEC-Busses. Nur in seltenen Fällen, z.B. zum Lesen einer Disketten-Fehlermeldung, muß dies vom Programm selbst übernommen werden. Beim Arbeiten mit Dateien sollte zum Übersenden des Namens immer die Betriebssystem-Routine verwendet werden. Zum Lesen eines Diskstatus ist kein Dateiname erforderlich, womit diese Routinen auch entfallen können.

Zu einer Datenübertragung sendet der Computer zuerst die Geräteadresse des gewünschten Gerätes. Das angesprochene Gerät analysiert dann als einziges alle weiteren Informationen. Ein Gerät kann als Empfänger oder Sender von Daten auftreten. Entsprechend wird die eine Betriebsart LISTEN (Hören) und die andere TALK (Sprechen) genannt. Welche Übermittlungsrichtung gewünscht wird, wird mit der Geräteadresse übersandt.

Die beiden Routinen heißen ebenfalls LISTEN und TALK. Im Akkumulator muß die Geräteadresse übergeben werden. Zur Datenübertragung muß dem Gerät auch eine Sekundäradresse mitgeteilt werden, damit z.B. ein Diskettenlaufwerk weiß, aus welcher Datei Daten gelesen werden sollen. Die Se-

kundäradresse wird mit den Routinen LSTNSA (nach LISTEN) und TALKSA (nach TALK) übersandt. Die Sekundäradresse, erhöht um 96 (Bit 5 und 6 gesetzt), wird im Akkumulator übergeben.

Das Lesen (bei TALK) und Schreiben (bei LISTEN) von Daten erfolgt dann mit den Routinen IECIN und IECOUT.

Am Ende der Übertragung muß die Verbindung mit UNLSTN bzw. UNTALK geschlossen werden. Ein Parameter muß nicht übergeben werden. Als Beispiel wird das Diskettenlaufwerk 8 mit dem Befehl "I0" initialisiert.

```
LISTEN   =   $FFB1
LSTNSA   =   $FF93
IECOUT   =   $FFA8
UNLSTN   =   $FFAE
         LDA  #8              ;GERAETEADRESSE=8
         JSR  LISTEN          ;LISTEN SENDEN
         LDA  #111            ;SEKUNDAERADRESSE=15+96=111
         JSR  LSTNSA          ;SEKUNDAERADRESSE SENDEN
         LDA  #"I"            ;BEFEHL SENDEN
         JSR  IECOUT
         LDA  #"0"
         JSR  IECOUT
         JMP  UNLSTN          ;UNLISTEN SENDEN + PROGRAMMENDE
```

Als Grafik sehen Sie in Abb. 9.3, wie die Datenübertragung über den IEC-Bus beim Lesen einer Datei ablaufen müßte.

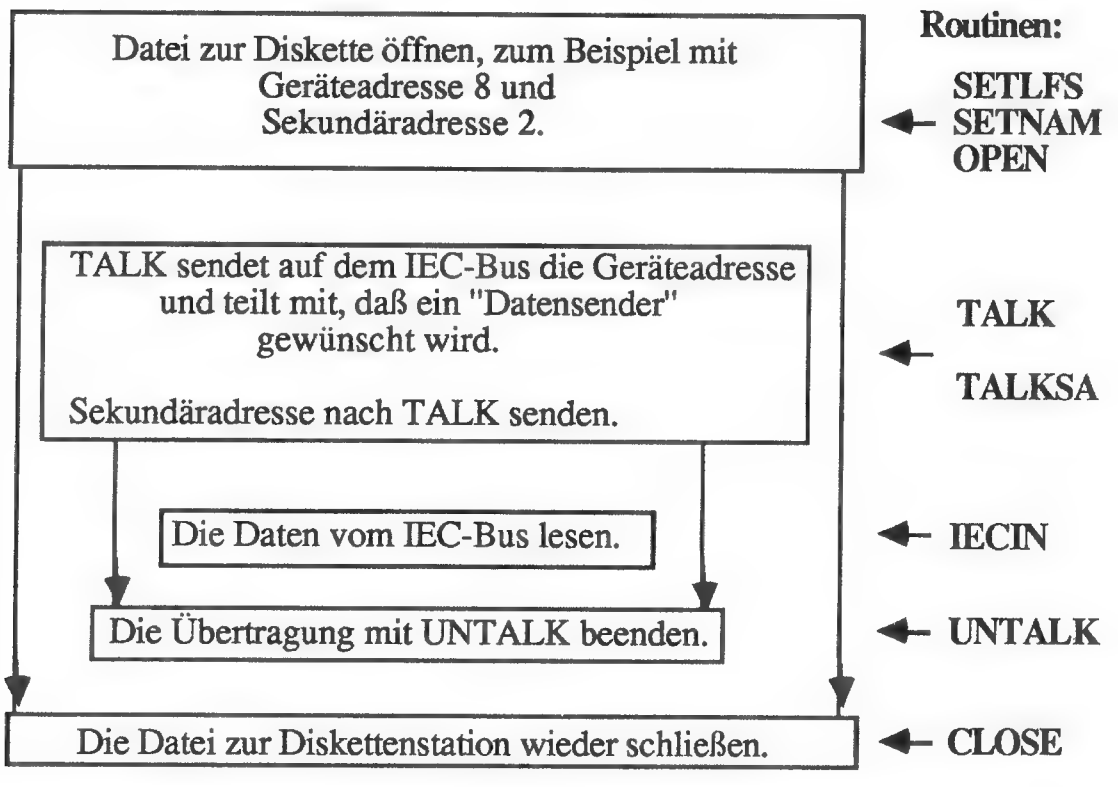


Abb. 9.3: Ablaufdiagramm der Datenübertragung über den IEC Bus

Kapitel 10

BASIC und Maschinensprache

Ein Assemblerprogramm gewinnt erst an Wert durch eine sinnvolle Anwendung, z.B. zur Unterstützung eines BASIC-Programms. In den bisherigen Kapiteln haben Sie das Programm im Editor eingetippt und mit dem Assembler übersetzen lassen. Zum Erstellen des Programms ist dies der beste, zum eigentlichen Verwenden des fertigen Programms aber der denkbar ungeeignetste Weg.

Es wäre günstig, wenn nur noch der Objektcode in den Speicher geschrieben werden müßte. Bei Maschinenprogrammen, die ein BASIC-Programm unterstützen sollen, ist es noch besser, wenn das BASIC-Programm selbst den Objektcode enthält und ihn in den Speicher schreibt.

Das Maschinenprogramm in den Speicher übertragen

Der Assembler erzeugt aus dem Assemblerprogramm den Objektcode, der das eigentliche Maschinenprogramm darstellt. Dieser wird ebenfalls vom Assembler in den Speicher geschrieben. Genauso könnte der BASIC-Befehl POKE die einzelnen Zahlenwerte in den Speicher schreiben. Dies ist der Ausgangspunkt für die weiteren Überlegungen. Zunächst ein Ausschnitt aus einem Assemblerlisting:

```
F1300   A9 21      0004         LDA     #"!"
F1302   20 D2 FF   0005         JSR     $FFD2
F1305   20 D2 FF   0006         JSR     $FFD2
F1308   A9 0D      0007         LDA     #13
F130A   4C D2 FF   0008         JMP     $FFD2
```

Um die Zahlenwerte in das BASIC-Programm eingeben zu können, müssen die Hexadezimalzahlen in Dezimalzahlen umgewandelt werden. Diese lauten dann wie folgt:

```
169 33 32 210 255 32 210 255 169 13 76 210 255
```

Mit 13 einzelnen POKE-Anweisungen wird das Maschinenprogramm in den Speicher geschrieben:

```
POKE  4864,169
POKE  4865,33
POKE  4866,32
  .
  .
POKE  4875,210
POKE  4876,255
```

Jedem BASIC-Programmierer wird beim Anblick dieser POKE-Liste klar, daß es viel einfacher geht. Die einzelnen Zahlenwerte können in einer DATA-Tabelle zusammengefaßt, in einer Schleife mit READ einzeln ausgelesen und mit POKE geschrieben werden. Die POKE-Anweisung benötigt eine Angabe, in welche Speicherbank die Werte geschrieben werden sollen. Der BASIC-Befehl BANK wählt den entsprechenden Speicher aus.

Das komplette BASIC-Programm sieht dann wie folgt aus:

```
10  BANK 15
20  FOR I=4864 TO 4876
30  READ X: POKE I,X
40  NEXT I
50  DATA 169,33,32,210,255,32,210,255,169,13,76,210,255
```

Das Maschinenprogramm wird in den Speicher übertragen und kann dort gestartet werden. Ein Assembler ist nicht mehr nötig.

Umständlich ist jetzt nur noch die Umwandlung der Hexadezimalwerte in Dezimalzahlen. Mit Hilfe einer Schleife könnten die einzelnen Werte des Maschinenprogramms direkt aus dem Speicher gelesen werden. Der Assembler schreibt den Objektcode in den Speicher und die BASIC-Funktion PEEK liest diesen wieder von dort. Als Ergebnis erhält man eine Liste von dezimalen Werten, die einzelnen Maschinenbefehle. Der langwierige Umwandlungsprozeß der Hexadezimalzahlen entfällt. Als Beispiel wird das obige Maschinenprogramm aus dem Speicher gelesen:

```
FOR I=4864 TO 4876 : PRINT PEEK(I); : NEXT I
169 33 32 210 255 32 210 255 169 13 76 210 255
```

Der beschriebene Weg, ein Maschinenprogramm in den Speicher zu schreiben, hat aber auch einen Nachteil. Sobald das Programm länger als ca. 1.5 KByte wird, ist die Tipparbeit für die DATA-Liste sehr groß. Dann benötigt auch die Schleife zum Schreiben des Programms bereits eine deutlich spürbare Zeit. Stellen Sie sich nur vor, Sie müßten für ein Programm 2000 Werte eingeben und dies auch noch ohne Tippfehler! Das BASIC-Programm benötigt schon ca. 1 Minute, um diese Anzahl von Werten aus der DATA-Tabelle zu lesen und in den Speicher zu schreiben. Für Programme dieser Länge muß ein anderer Weg beschritten werden.

Es gibt eine spezielle Lade-Anweisung BLOAD, die eine Programmdatei an eine durch die Datei selbst bestimmte Adresse lädt. Die normalen Lade-Befehle LOAD und DLOAD machen ja auch nichts anderes. Bei diesen wird jedoch immer an die BASIC-Startadresse geladen.

Die einzelnen Zahlenwerte des Maschinenprogramms werden in eine Datei geschrieben und im BASIC-Programm mit BLOAD wieder geladen.

Es gibt zwei Wege, das Maschinenprogramm in eine Datei zu schreiben. Eine entsprechende BASIC-Anweisung BSAVE liest die Speicherinhalte und schreibt sie in eine Datei:

```
BSAVE "BEISPIEL.O",ON B0,P4864 TO 4876
```

Der Dateiname wurde mit dem Kennbuchstaben ".O" versehen, damit im Inhaltsverzeichnis sofort erkannt wird, daß es sich hier um einen gespeicherten Objektcode handelt. Die Angabe "ON B0" teilt dem Befehl mit, daß sich der Speicherbereich im Bank 0 befindet. Der Bereich selbst wird mit den Adreßangaben "P4864 TO 4876" bestimmt.

Der zweite Weg, um das Maschinenprogramm in eine Datei zu schreiben, besteht gleich beim Assemblieren. Normalerweise beauftragt die Anweisung .OBJ M den Assembler, den erzeugten Objektcode in den Speicher zu schreiben. Andere Ziele sind jedoch auch erlaubt. Die Befehlsform .OBJ "BEISPIEL" leitet den Objektcode direkt in eine Diskettendatei. Der Assembler fügt selbständig das Kürzel ".O" an. Im Inhaltsverzeichnis taucht die Datei deshalb auch mit dem Namen "BEISPIEL.O" auf. Ein Programmkopf müßte dann wie folgt aussehen:

```
*=    $1300
      .OBJ  "BEISPIEL"
         .
         .
```

Achtung: Der Objektcode wird jetzt nur in die Diskettendatei geschrieben und überhaupt nicht mehr in den Speicher. Das Programm kann also nicht mehr einfach mit !GO $1300 gestartet werden.

Das BASIC-Programm zum Laden des Objektcodes, egal auf welchem Weg gespeichert wurde, enthält jetzt nur eine Zeile:

```
10 BLOAD "DATEINAME.O",ON B0
```

Auch hier erfolgt wieder eine Vorgabe der Speicherbank mit "ON B0". Diese Angabe kann auch entfallen, da Bank 0 vorangestellt ist. Wird aber z.B. Objektcode in Bank 1 geladen, ist eine Bankangabe, in diesem Fall "ON B1", unbedingt erforderlich.

Starten eines Maschinenprogramms

Zum Aufruf von Maschinenprogrammen dient die spezielle BASIC-Anweisung SYS. Als Parameter folgt ihr die Startadresse des Programms. Ein komplettes BASIC-Programm mit Übertragen des Objektcodes und mit Starten des Programms sieht wie folgt aus:

```
10 BANK 15
20 FOR I=4864 TO 4876
30 READ X : POKE I,X
40 NEXT I
50 SYS 4864
60 DATA 169,33,32,210,255,32,210,255,169,13,76,210,255
```

Geben Sie das Programm ein, und starten Sie es. Auf dem Bildschirm erscheinen, wie Sie aus dem Programm sicher schon erkannt haben, zwei Ausrufezeichen.

Parameterübergabe an das Maschinenprogramm

Bei komplexeren Maschinenprogrammen müssen möglicherweise Parameter übergeben werden. Die Werte werden z.B. in Speicherzellen der Zero-Page geschrieben. Ein Maschinenprogramm kann sie von dort lesen und verarbeiten. Auch die Rückgabe von Werten ist auf diesem Weg möglich.

Das Beispielprogramm aus den letzten Abschnitten soll abgewandelt werden. In der Speicherzelle 250 soll diesem mitgeteilt werden, welches Zeichen zweimal ausgegeben werden soll. Das Assemblerprogramm sieht dann wie folgt aus:

```
LDA   250
JSR   $FFD2
JSR   $FFD2
LDA   #13
JMP   $FFD2
```

Ein BASIC-Programm könnte diese Routine wie folgt enthalten und einsetzen:

```
10 BANK 15
20 FOR I=4864 TO 4876
30 READ X : POKE I,X
40 NEXT I
50 POKE 250,42: SYS 4864
60 POKE 250,46: SYS 4864
70 DATA 165,250,32,210,255,32,210,255,169,13,76,210,255
```

Das BASIC-Programm schreibt das Maschinenprogramm in den Speicher. Beim ersten Aufruf wird als Parameter der ASCII-Code eines Sterns übergeben und beim zweiten Mal der Code eines Dezimalpunktes.

Auf demselben Weg könnte das Maschinenprogramm Werte an das BASIC-Programm zurückgeben. Das Beispielprogramm könnte z.B. mitteilen, auf welchem Gerät die Ausgabe erfolgte, oder ob bei der Ausgabe ein Fehler auftrat.

Der BASIC-Interpreter des C128 bietet noch eine weitere Methode der Parameterübergabe des Maschinenprogramms an das BASIC-Programm. Die Registerinhalte können nach Beendingung des Maschinenprogramms Variablen zugewiesen werden. Die Namen dieser Variablen werden mit dem Befehl RREG bestimmt.

RREG	AC,XR,YR,SR	AC enthält den Akkumulator XR enthält das X-Register YR enthält das Y-Register SR enthält das Statusregister
RREG	TEST,AB	TEST enthält den Akkumulator AB enthält das X-Register

Ein Beispielprogramm soll die Inhalte der Speicherzellen 250 und 251 addieren und das Ergebnis im Akkumulator an das BASIC-Programm zurückgeben. Der Text des Assemblerprogramms lautet wie folgt:

```
LDA   250
CLC
ADC   251
RTS
```

Ein komplettes BASIC-Programm mit Schreiben des Maschinenprogramms würde entsprechend wie folgt lauten:

```
10 BANK 15
20 FOR I=4864 TO 4869
30 READ X : POKE I,X
40 NEXT I
50 RREG SUM
60 POKE 250,100: POKE 251,45: SYS 4864: ?SUM
70 DATA 165,250,24,101,251,96
```

Gleitkommazahlen

Es gibt noch einen dritten Weg, sowohl an das Maschinenprogramm als auch umgekehrt Parameter zu übergeben. Zuvor müssen jedoch einige Begriffe geklärt werden. Als in Kapitel 6 Arithmetik in Assemblerprogrammen behandelt wurde, haben Sie sich sicher die Frage gestellt, wie es möglich ist, daß in BA-

SIC-Programmen mit Zahlenwerten bis 10^{38} gerechnet werden kann. Die Beispielprogramme schafften gerade Zahlen bis 65535. Gemäß der Erweiterung der Zahlenwerte von 255 auf 65535 durch Koppelung mehrerer Bytes könnte man dies für größere Zahlen weiter fortsetzen. Mit drei Bytes auf 16777215 oder mit zehn Bytes auf ca. 10^{24}. Aber bereits 10 Bytes nehmen zu viel Speicher weg, und es sinkt auch die Verarbeitungsgeschwindigkeit. Um das Problem zu lösen, greift man zu einem Trick.

Bestimmt haben Sie auf Taschenrechnern und bei Zahlausgaben des C128 schon die sogenannte Zehner-Exponent-Darstellung gesehen. Die Zahl wird aus einem Dezimalbruch (Mantisse) und einem Exponenten zur Basis 10 zusammengesetzt.

$$\begin{aligned} 453245 &= 4.53245 * 10^{5} \\ 100 &= 1.0 * 10^{2} \\ 0.0031 &= 3.1 * 10^{-3} \end{aligned}$$

Um mit größeren Binärzahlen zu rechnen, wird eine ganz ähnliche Darstellung gewählt. Da bei Computern nur mit Binärzahlen gerechnet wird, handelt es sich bei der Mantisse entsprechend um eine Binärzahl und einen Exponenten zur Basis 2. Zahlen würden in dieser Darstellung wie folgt aussehen:

$$0.10010011 * 2^{10101}$$
$$0.11101011 * 2^{111}$$

Streng genommen müßte man für den Exponenten jetzt auch die binäre Darstellung 10 (=2) schreiben. Zur Unterscheidung vom Zehner-Exponent wird jedoch diese Darstellung gewählt. An den Beispielen sehen Sie auch, daß es ohne weiteres möglich ist, binäre Brüche niederzuschreiben.

Der C128 hat in den ersten 256 Bytes sechs Speicherzellen zu einem sogenannten Floating-Point-Accumulator (Fließkomma-Akkumulator) oder kurz FAC zusammengefaßt. In diesen Speicherzellen laufen alle Arithmetikaufgaben mit Zahlen ab.

Die Ziffern jeder Zahl werden zur Darstellung der Mantisse in ein Format der Form 0.1.... gebracht. Die höchstwertige Stelle der Mantisse muß also immer den Wert Eins haben. Die erste Stelle nach dem Komma hat den Stellenwert 1/2. Folglich hat jede Mantisse einen Wert zwischen 0.5 (binär: 0.1) und 1 (binär: <1.0 oder 0.1111...). Die Mantisse wird in vier Bytes gespeichert. Alle Stellen, die nicht benutzt werden, werden mit Nullen aufgefüllt.

Der Exponent wird in einer weiteren Speicherzelle abgelegt. Bei Zahlen größer 0.5 ist die Mantisse kleiner als die eigentliche Zahl. Der Exponent muß

in diesem Fall einen positiven Wert haben (z.B $2^2 = 4$, $2^7 = 128$). Bei Zahlen kleiner als 0.5 hat der Exponent entsprechend einen negativen Wert. Hier ist die Mantisse größer als die Zahl (z.B. $2^{-1} = 0.5$, $2^{-5} = 0.03125$). Zur Darstellung des Exponenten wird zu seinem Wert 128 addiert.

Die Werte von 0 bis 127 stellen also einen negativen Exponenten dar und die Zahlen von 128 bis 255 einen positiven. Zu beachten ist, daß diese Darstellung nicht mit der der Zweierkomplement-Arithmetik übereinstimmt.

Als letzte Information über die Zahl fehlt noch das Vorzeichen. Das Bit 7 einer weiteren Speicherzelle bestimmt, ob es sich um eine positive (Bit 7 = 0) oder eine negative (Bit 7 = 1) handelt. Enthält das Vorzeichenbyte den Wert Null, handelt es sich bei der gesamten Zahl auch um Null.

Der Wert der Mantisse spielt keine Rolle. Auch ist es dadurch nicht möglich, Zahlen wie –0 oder +0 darzustellen. Jetzt ein paar Beispiele für Zahlenumwandlungen:

$1000_{(dez)} = 11\ 1110\ 1000$
$= 0.1111\ 1010\ 00 * 2^{1010}$

138 250 000 000 000 1
↑ 4 Bytes Mantisse ↑
Exponent (138 = 10 + 128) Vorzeichen (Bit 7=0 ⇒ positiv)

$-0.375 = -0.011$
$= -0.11 * 2^1$

127 192 000 000 000 128
↑ 4 Bytes Mantisse ↑
Exponent (127 = –1 + 128) Vorzeichen (Bit 7=1 ⇒ negativ)

Im Speicher stehen die einzelnen Bestandteile der Zahl in den folgenden Speicherzellen:

99	100 101 102 103	104
Exp.	-- Mantisse ---	Vorz.

Zur Übung der ganze Umwandlungsprozeß rückwärts:

```
133    170 064 000 000    001
↑                         ↑
Exp.=133 – 128            Zahl ist positiv
```

⇒ $0.1010101001 * 2^5$
= 10101.01001
= 21.28125

Es kann jetzt auch die größte darstellbare Zahl errechnet werden. Die Mantisse hat den maximalen Wert 0.999999... (binär: 0.111111...) und der Exponent den maximalen Wert 127 (127+128=255). Folglich ist der größte darstellbare Wert $0.99999... * 2^{127}$, was etwa $2^{127} = 1.7014 * 10^{38}$ entspricht. Aus dem Handbuch zum C128 ist Ihnen dieser Wert sicher bekannt. Jetzt wissen Sie auch, wie er zustande kommt.

Sie werden sich wahrscheinlich fragen, wozu diese ganzen Erklärungen dienen? Ein eigenes Maschinenprogramm könnte niemals direkt mit dieser Zahldarstellung arbeiten. Der Aufwand hierzu wäre viel zu groß. Jedoch stellt das ROM des BASIC-Interpreters viele Routinen zur Verfügung, um Fließkomma-Zahlen zu verarbeiten.

Die USR-Funktion

Wie oben bereits erwähnt, stellt der BASIC-Interpreter einen weiteren Weg zur Verfügung, Parameter zu übergeben. Die BASIC-Funktion USR übernimmt diese Aufgabe. Das in Klammern eingeschlossene Argument der Funktion wird im FAC dem Maschinenprogramm mitgeteilt, und das Maschinenprogramm gibt seinerseits im FAC den errechneten Wert zurück. Die Startadresse des Maschinenprogramms wird nicht wie beim SYS-Befehl in der Funktion genannt, sondern muß zuvor in zwei Speicherzellen abgelegt werden.

Diese haben die Adressen 4633 ($1219) und 4634 ($121A). Wie bei allen Adressen wird auch hier in der niedrigeren Speicherzelle (4633) das LO-Byte und in der höheren (4634) das HI-Byte gespeichert.

Ein erstes Beispiel soll die Speicherzellen des FAC auslesen und auf dem Bildschirm darstellen. Sie können dadurch mit der Fließkomma-Darstellung nochmals selbst arbeiten. Das Assemblerlisting besteht nur aus einer Schleife, die den FAC in einen anderen Speicherbereich kopiert. Dies ist notwendig, da z.B. die BASIC-Funktion PEEK, die zum Auslesen der Speicherinhalte notwendig ist, den Inhalt des FAC sofort wieder verändern würde.

```
FAC         =     99
COPY        =     250
            LDX   #0                ;SCHLEIFENZAEHLER STARTEN
SCHLEIFE    LDA   FAC,X             ;EIN FAC-BYTE LESEN
            STA   COPY,X            ;EINE KOPIE ANFERTIGEN
            INX                     ;DEN SCHLEIFENZAEHLER VERMINDERN
            CPX   #6                ;MIT DEM ENDWERT VERGLEICHEN
            BNE   SCHLEIFE          ;ALLE 6 BYTES ?,NEIN =>
            RTS                     ;DAS PROGRAMM VERLASSEN
```

Das BASIC-Programm legt die Startadresse ($1300) des Programms fest und ruft es mit der USR-Funktion auf. Anschließend werden die einzelnen Komponenten des FAC ausgegeben.

```
10 BANK 0 : POKE 4633,0: POKE 4634,19
20 INPUT "ZAHL";Z
30 X=USR(Z)
40 PRINT "EXPONENT: ";PEEK(250)
50 PRINT "MANTISSE: ";
60 FOR I=251 TO 254: PRINT PEEK(I); : NEXT I
70 PRINT : PRINT "VORZEICHEN: ";PEEK(255)
```

Für echte Problemlösungen hat die USR-Funktion noch einen Nachteil. Die Fließkomma-Darstellung der Zahlen kann ohne großen Aufwand nicht direkt verarbeitet werden. Hier hilft eine ROM-Routine ab der Adresse $849F (33951) weiter. Diese wandelt eine Fließkomma-Zahl, die im FAC abgelegt ist, in eine Integer-Zahl um.

Alle Zahlen im Bereich von –32767 bis +32767 werden gemäß der Zweierkomplement-Darstellung transformiert und im Akkumulator (HI-Byte) und dem Y-Register (LO-Byte) an das aufrufende Programm zurückgegeben. Wie Zahlenwerte in dieser Form gehandhabt werden, haben Sie in den zurückliegenden Kapiteln gelernt.

Zu klären ist jetzt noch, wie Integer-Zahlen wieder in Fließkomma-Zahlen gewandelt werden, um sie wieder mit der USR-Funktion an das BASIC-Programm zurückgeben zu können. Auch für diesen Zweck gibt es wieder eine ROM-Routine. Dem Unterprogramm ab der Adresse $793C (31036) wird im Akkumulator (HI-Byte) und im Y-Register (LO-Byte) der Integer-Wert übergeben.

Entsprechend der Zweierkomplement-Darstellung werden hieraus Zahlen zwischen –32767 und +32767 gebildet und im FAC abgelegt. Als Beispiel soll ein Maschinenprogramm den übergebenen Wert mit 10 multiplizieren und das Ergebnis wieder an das BASIC-Programm zurückmelden. Methoden zur schnellen Multiplikation mit 10 haben Sie in Kapitel 6 kennengelernt. Jetzt muß nur mit einer 16 Bit breiten Zahl gearbeitet werden.

```
ZAHL    =   250
FACINT  =   $849F
INTFAC  =   $793C
        JSR FACINT      ;FAC NACH INT WANDELN
        STY ZAHL        ;DAS LO-BYTE SPEICHERN
        STA ZAHL+1      ;DAS HI-BYTE SPEICHERN
        TAX             ;DAS HI-BYTE IM X-REG. MERKEN
        ASL ZAHL        ;DIE ZAHL * 4
        ROL ZAHL+1
        ASL ZAHL        ;DIE ZAHL * 2 (* 4)
        ROL ZAHL+1
        TYA             ;DAS LO-BYTE HOLEN
        CLC
        ADC ZAHL        ;UND ZUR ZAHL ADDIEREN
        STA ZAHL        ;UND DAS ERGEBNIS SPEICHERN
        TXA             ;DAS HI-BYTE HOLEN
        ADC ZAHL+1      ;UND ZUR ZAHL ADDIEREN (* 5)
        STA ZAHL+1      ;UND DAS ERGEBNIS SPEICHERN
        ASL ZAHL        ;DIE ZAHL * 2 (* 10)
        ROL ZAHL+1
        LDY ZAHL        ;DAS LO-BYTE LADEN
        LDA ZAHL+1      ;DAS HI-BYTE LADEN
        JMP INTFAC      ;INT-ERGEBNIS IN FAC BRINGEN
```

Zum Programm ein paar kurze Erklärungen: Durch die Routine FACINT wird der Inhalt des Gleitkomma-Akkumulators (FAC) in eine Integerzahl verwandelt, die im Akkumulator und Y-Register übergeben wird. Diese Zahl wird sowohl im Speicher abgelegt als auch in den Registern X (HI-Byte) und Y (das LO-Byte befindet sich bereits dort) zwischengespeichert. Im nächsten Programmteil wird die im Speicher abgelegte Zahl durch Verschieben nach links mit 4 multipliziert. Im folgenden muß der ursprüngliche Wert der Zahl wieder addiert werden.

Jetzt wird auch klar, warum die Zahl zweimal gespeichert werden mußte, einmal, um ihren Wert zu bearbeiten, und einmal, um den Wert für einen späteren Zeitpunkt wieder parat zu haben. Nach dieser Addition ist die Zahl mit 5 multipliziert.

Durch ein weiteres Verschieben nach links wird schließlich das gewünschte Ziel, die Multiplikation mit 10, erreicht. Das Ergebnis liegt jetzt in den Speicherzellen ZAHL und ZAHL+1 vor. Die Funktion USR, mit der diese Routine aufgerufen werden soll, verlangt als Rückgabewert eine Zahl im FAC. Für diesen Zweck wird das Ergebnis in den Akkumulator (HI-Byte) und das Y-Register (LO-Byte) geladen und mit Aufruf der Routine INTFAC im FAC abgelegt.

Die im Programm formulierte Lösung ist sehr einfach gehalten, damit sie leicht verständlich bleibt. Durch Optimierungen läßt sich die Multiplikation kürzen und damit auch beschleunigen. Die ganzen Transfers und Manipulatio-

nen von Daten lassen sich jedoch dann nicht mehr so leicht überschauen. Wenn Sie Lust haben, versuchen Sie doch einmal, das Programm zu verbessern.

Jetzt wieder zurück zum eigentlichen Thema dieses Kapitels, der Verbindung von BASIC- und Assemblerprogrammen. Nach dem Assemblieren werden die einzelnen Werte des Objektcodes in DATA-Zeilen zusammengefaßt und von einer Schleife in den Speicher geschrieben. Das Maschinenprogramm soll über die Funktion USR aufgerufen werden. Dieser wird über die Speicherzellen 4633 ($1219) und 4634 ($121A) die Startadresse (hier 4864) des Programms mitgeteilt. Im BASIC-Programm erfolgt dies durch zwei POKE-Anweisungen. Das komplette BASIC-Programm hat dann folgendes Aussehen:

```
10 FOR I=4864 TO 4901
20 READ X : POKE I,X
30 NEXT I
40 POKE 4633,0 : POKE 4634,19
50 DATA 32,159,132,132,250,133,251,170,6,250,38,251,6,250
60 DATA 38,251,152,24,101,250,133,250,138,101,251,133,251
70 DATA 6,250,38,251,164,250,165,251,76,60,121
```

Nach dem Starten des BASIC-Programms kennt der C128 die Lage des Programms für die Funktion USR, und damit kann diese benutzt werden. Hier ein paar Beispiele:

```
? USR(10)
100

? USR(34 + 76)
1100

? USR(- 121)
- 1210

? 10 + USR(2)
30

A=USR(745)/2 + 5
? A
3730
```

Wie aus einem Beispiel zu ersehen ist, dürfen auch negative Werte übergeben werden. Die Routine FACINT erwartet als Eingangswert im FAC Zahlen von –32767 bis +32767. Diese werden gemäß der Zweierkomplement-Darstellung in Integer-Zahlen umgewandelt. Aufgrund der Vorteile der Zweierkomplement-Darstellung liefert jetzt dieselbe Routine korrekte Ergebnisse für positive und negative Zahlen.

Wie schon bei vielen Beispielen in den letzten Kapiteln bemerkt, funktioniert die Routine nur, solange die vorgegebenen Zahlgrenzen nicht überschritten werden, d.h. positive Ergebnisse nicht größer als 32767 bzw. negative nicht kleiner als –32767 werden. Sollten diese Grenzen jedoch einmal überschritten werden, entsteht ein unsinniges Ergebnis. Hier zwei Beispiele für solche Fälle:

```
? USR(- 3782)
27716

? USR(4523)
- 20306
```

Es soll noch auf zwei weitere Routinen zur Zahlumwandlung hingewiesen werden. Die Routine FACADR ab der Adresse $8815 (34837) erwartet als Inhalt des FAC Werte von 0 bis 65535. Die Zweierkomplement-Darstellung wird hier also ignoriert. Die entsprechende Routine ADRFAC zur Übergabe der Zahlen in den FAC liegt im ROM ab der Adresse $84C9 (33813). Wie bei den bereits aufgezählten Routinen verwenden auch diese beiden den Akkumulator und das Y-Register zur Übergabe der Werte.

Mit diesen vier hier genannten Routinen haben Sie die nötigen Hilfsmittel an der Hand, um einzelne spezielle Aufgaben zu lösen. Die Gleitkomma-Zahlen in Verbindung mit der USR-Funktion bieten noch viele weitere Möglichkeiten. Im ROM des C128 sind Routinen integriert, die Gleitkomma-Zahlen miteinander addieren, subtrahieren, multiplizieren usw. Auch das Ziehen der Quadratwurzel, das Logarithmieren sowie die Bildung des Sinuswertes ist mit diesen Routinen notwendig.

Um diese jedoch anwenden zu können, sind tiefergehende Kenntnisse über die Behandlung von Gleitkomma-Zahlen und über das ROM des C128 nötig. Die entsprechenden Erklärungen würden selbst wieder mehrere Kapitel füllen. Es wird deshalb hier darauf verzichtet.

Speicherbereiche für das Maschinenprogramm

Wie jedes Programm benötigt auch ein Assemblerprogramm, wenn es assembliert wurde und als Maschinenprogramm vorliegt, Speicher. Die bisherigen Beispiele waren immer recht kurz, und es traten nie Probleme auf. Zu welchen Hilfsmitteln kann man aber greifen, wenn Programme Längen von 1 KByte und mehr erlangen?

Im C128 ist bereits ein Speicherbereich für eigene Anwendungen freigehalten worden. Dieser liegt in Bank 0 im Adreßbereich $1300 (4864) bis $1BFF

(7167). Jetzt wird Ihnen auch verständlich, warum bei Beispielen immer die Adresse $1300 als Startadresse gewählt wurde. Der Assembler EDASS belegt von diesem freien Speicher leider 512 Bytes, nämlich den Speicher ab der Adresse $1A00 (6656), womit nur noch der Bereich $1300 bis $19FF zur Verfügung übrig bleibt. Aus diesen beiden Grenzen errechnet sich ein verfügbarer Speicher von 1792 Bytes bzw. ohne EDASS 2304 Bytes.

Bei längeren Programmen gibt es mehrere andere Speicherpositionen, die mehr Platz bieten. Entweder werden Teile des für BASIC-Programme zur Verfügung stehenden Speichers benutzt oder Teile des Speichers für BASIC-Variablen. Genau diesen Weg beschreitet z.B. EDASS auch. Im Speicherbereich für BASIC-Programme sind die Labels gespeichert und im Speicherbereich für BASIC-Variablen die Programmtexte und das eigentliche EDASS-Programm. Aber keine Angst, diese Belegung behindert Sie nicht, nochmals Speicher für Ihre Anwendungen zu reservieren.

Zuerst zum BASIC-Programm-Speicher, der ab der Adresse $1C00 (7168) in Bank 0 beginnt. Im einfachsten Fall kann das Maschinenprogramm anstatt eines BASIC-Programms in den Speicher geschrieben werden. Ein BASIC-Programm kann dann natürlich nicht mehr im Speicher stehen.

So unsinnig die Methode klingt, ist sie aber in leicht abgewandelter Form sehr häufig zu finden. Vor dem Maschinenprogramm steht im Speicher eine einzelne BASIC-Programmzeile, die den Aufruf des Maschinenprogramms enthält. Die BASIC-Zeile und das Maschinenprogramm werden zusammen in einer Programmdatei gespeichert und folglich beim Laden auch gemeinsam gelesen. Solche Programme erkennen Sie sofort mit dem LIST-Befehl. Als Programmtext zeigt sich z.B. nur "10 SYS 7210". Laden Sie doch einfach einmal die Programmdatei "EDASS 128", und betrachten Sie sich den Inhalt des BASIC-Programms.

Oft wird ein Assemblerprogramm jedoch nur zur Unterstützung eines BASIC-Programms herangezogen. Die zuvor beschriebene Methode ist dann ungeeignet. In solchen Fällen wird der zur Verfügung stehende Speicher für das BASIC-Programm gekürzt. In den Speicherzellen $1212 (4626, LO-Byte) und $1213 (4627, HI-Byte) in Bank 0 wird festgelegt, bis zu welcher Adresse der BASIC-Programmspeicher verfügbar ist.

Nach dem Einschalten des C128 wird hier der Normalwert $FF00 (65280) eingetragen. Durch entsprechende POKE-Anweisungen kann diese Grenze herabgesetzt werden, womit der Bereich ab der eingetragenen Grenze bis zur Adresse $FF00 für ein BASIC-Programm gesperrt ist. Ein Maschinenprogramm kann diesen Speicher nutzen. Ist EDASS gleichzeitig geladen, muß beachtet werden, daß der Speicher zwischen einem BASIC-Programm und den

EDASS-Labels aufgeteilt ist. Um trotzdem Speicher zu gewinnen, sind folgende Schritte auszuführen:

1. Durch Abfragen mit PEEK aus den Speicherzellen $1212 und $1213 wird die derzeitige Grenze für ein BASIC-Programm und damit der aktuelle Anfang für die Labels des Assemblers festgestellt.

2. Durch POKE-Anweisungen wird die Speichergrenze herabgesetzt.

3. Ein Maschinenprogramm kann den Speicher zwischen der neuen, durch POKE-Anweisungen festgelegten und der alten, durch die PEEK-Funktion ausgelesenen Grenze belegen.

Auf eine ähnliche Weise kann der Speicher der BASIC-Variablen vermindert werden. Im Gegensatz zum BASIC-Programmspeicher werden die Variablen in Bank 1 gespeichert. Die Speicherzellen $2F (47, LO-Byte) und $30 (48, HI-Byte) bestimmen die Anfangsadresse der BASIC-Variablen. Der Normalwert liegt bei $400 (1024). Ein niedrigerer Wert darf hier niemals eingetragen werden. Durch POKE-Anweisungen kann diese Grenze auf eine höhere Adresse verlegt werden.

Der Bereich zwischen $400 und dieser neuen Adresse ist dann für ein Maschinenprogramm frei verfügbar. Entsprechend kann auch die Obergrenze verändert werden. Diese wird in den Speicherzellen $39 (57, LO-Byte) und $3A (58, HI-Byte) gespeichert. Diese sind mit dem Normalwert $FF00 belegt. Durch Verändern wird der Bereich bis zu dieser Maximaladresse $FF00 für ein Maschinenprogramm verfügbar.

Bei gleichzeitigem Betrieb von EDASS ist zu beachten, daß der größte Teil des Variablenspeichers für die Programmtexte des Assemblers reserviert ist. Außerdem beginnt ab der Adresse $B400 das EDASS-Programm. Wie beim BASIC-Programmspeicher muß auch hier zuerst die aktuelle Endadresse der BASIC-Variablen und damit die Anfangsadresse der Assemblertexte ausgelesen werden. Erst mit dieser Grenze kann entschieden werden, durch welche neue Adresse der Variablenspeicher begrenzt werden soll. Auch wird damit erst die Grenze des frei verfügbaren Bereiches ausfindig gemacht.

Nach Veränderungen an den Grenzen des Variablenspeichers muß dieser mit dem BASIC-Befehl CLR neu initialisiert werden. Es werden damit einige weitere Adressen zur Verwaltung der Variablen auf den korrekten Wert gesetzt und, was noch viel wichtiger ist, es werden die Reste alter Variablen als ungültig definiert. Im zweiten Teil des Buches, in dem Sie mit allen Befehlen von EDASS vertraut gemacht werden, lernen Sie auch einen Befehl kennen, mit dem Sie die Aufteilung des Speichers zwischen BASIC und EDASS neu bestimmen können. Damit läßt sich der EDASS-Datenspeicher reduzieren, und es kann großzügig Speicher für eigene Programme abgespalten werden.

Kapitel 11

Der Maschinensprache-Monitor

Die Funktion eines Maschinensprache-Monitors, kurz Monitor, läßt sich am besten beim Versuch erklären, den Namen "Monitor" zu deuten. Mit derartigen Programmen betrachtet der Benutzer wie mit einer Video-Überwachungsanlage den Speicher. Er kann den Speicher in verschiedenen Darstellungen anschauen, diesen verändern, auf Diskette speichern und wieder von dort einlesen.

Der C128 hat in seinen ROMs ein Monitorprogramm fest eingebaut. Dieses wird mit dem BASIC-Befehl MONITOR aufgerufen. Außerdem wird bei Erreichen des Befehls BRK, der, wie Ihnen aus Kapitel 8 bekannt ist, softwaremäßig einen Interrupt auslöst, der Monitor aufgerufen.

Auf die Bedeutung dieser Eigenschaft wird zu einem späteren Zeitpunkt eingegangen. Nach dem Aufruf des Monitors zeigt dieser diverse Informationen an, die zunächst unbeachtet bleiben. Hier trotzdem einmal das Aussehen:

```
MONITOR
   PC  SR AC XR YR SP
;FB000 00 00 00 00 F8
```

Der Cursor wartet am Zeilenanfang auf die Eingabe von Kommandos. Diese bestehen jeweils aus einem Buchstaben und meist aus mehreren nachfolgenden Parametern. Haben Sie bei der Eingabe einen Fehler gemacht, oder hat sich sonst bei der Ausführung ein Fehler ergeben, wird dies durch ein Fragezeichen (?) signalisiert.

Das Zeichen erscheint in der Spaltenposition, in der auch der Fehler auftrat. Konnte also z.B. das Monitorprogramm nur die halbe Befehlszeile analysieren, wird auch nach der halben Zeile ein Fragezeichen ausgegeben.

Zum Verlassen dient das Kommando "X". Geben Sie diesen Buchstaben ein, und drücken Sie danach die <RETURN>-Taste. Das Monitorprogramm wird verlassen, und durch die Meldung "READY." zeigt der C128, daß er wieder zur Aufnahme von BASIC-Eingaben bereit ist.

Den Speicher betrachten und verändern

Mit dem zweiten Kommando, das Sie jetzt kennenlernen, wird der Speicherinhalt mehrerer Speicherzellen auf dem Bildschirm dargestellt. Rufen Sie den Monitor auf, und geben Sie folgende Befehlszeile ein:

```
M 1000 1015
>01000 07 06 0A 07 06 0A 05 08:........
>01008 09 05 47 52 41 50 48 49:..GRAPHI
>01010 43 44 4C 4F 41 44 22 44:CDLOAD"D
```

Bei diesem Gewirr aus Zahlen und Buchstaben gibt es sicher viel zu erklären. In der Befehlszeile wird mit dem Buchstaben "M" die Ausgabe des Speicherinhalts befohlen. Welcher Bereich des Speichers erscheinen soll, bestimmen die zwei nachfolgenden Hexadezimalzahlen. Diese werden hier ohne Voranstellen eines Dollarzeichens eingegeben (wenn Sie es unbedingt wünschen, dürfen Sie das Dollarzeichen auch hier eingeben).

Im Klartext bedeutet die Befehlszeile "M 1000 1015": "Gib den Inhalt der Speicherzellen von $1000 bis $1015 auf dem Bildschirm aus". Das Ergebnis sehen Sie direkt unter dem Befehl.

Die einzelnen Speicherinhalte werden als Hexadezimalzahlen ausgegeben. 8 Werte im 40-Zeichen- bzw. im 80-Zeichen-Modus 16 Werte werden in einer Zeile zusammengefaßt. Die Adresse des ersten dargestellten Wertes wird am Zeilenanfang ausgedruckt.

Am Zeilenende sehen Sie zusätzlich die Zahlenwerte als Zeichen des ASCII-Codes ausgegeben. Der Monitor hebt diese Zeichen durch eine inverse Darstellung hervor. Im obigen Beispiel ist dies durch Fettdruck angedeutet. Sind z.B. im Speicher Texte abgelegt, können diese leicht erkannt und gelesen werden. Im ASCII-Code sind den Werten von 0–31 und von 128 –159 Steuerfunktionen zugeordnet. Da diese nicht als Zeichen dargestellt werden können, wird stellvertretend ein Dezimal-Punkt ausgegeben. Da immer 8 Werte in einer Zeile dargestellt werden, endet die Ausgabe im Beispiel auch nicht mit der Adresse $1015, sondern, wenn man nachrechnet, mit der Adresse $1017.

Noch ein Wort zu den Adressen am Zeilenanfang: Wie Sie sehen, bestehen diese aus fünf Ziffern. Die vorderste Ziffer stellt die gewählte Banknummer und die letzten vier Stellen die eigentliche Adresse dar. In der Befehlseingabe dürfen natürlich genauso fünf Ziffern verwendet werden, um die Speicherbank zu wählen. Die Wahl erfolgt hier genauso wie beim BASIC-Befehl BANK: 0 für Bank 0, 1 für Bank 1 und 15 bzw. hier F für die ROMs. Hier ein Beispiel für eine Bankwahl:

```
M F4000 F400F
>F4000 4C 23 40 4C 09 40 4C 4D:L#@L.@L
>F4008 A8 20 CC FF 20 7A 41 20:M__ _A
```

Wird der Speicherinhalt in der beschriebenen Form dargestellt, spricht man auch von sogenannten Hex-Dumps des Speichers. Derartige Hex-Dumps müssen nicht unbedingt von einem Monitor stammen, z.B. kann auch der Assembler EDASS den erzeugten Objektcode als Hex-Dump ausgeben. Beim ersten Beispiel handelt es sich um einen Ausschnitt aus der Funktionstastenbelegung. Zu sehen ist der Text "GRAPHIC", die Belegung der Taste <F1> und "DLOAD, die Belegung der Taste <F2>. Das Zeichen "D" am Ende der Darstellung bildet den Anfang des Wortes "DIRECTORY". Wie wird jetzt der Speicherinhalt geändert?

Das Befehlswort DLOAD soll zu dem Wort "LOAD" gekürzt werden. Die Speicherzelle, die das Zeichen "D" enthält, wird hierzu einfach mit dem Wert einer Leerstelle belegt. Der Cursor wird auf die Hexadezimaldarstellung des Wertes positioniert, wo der alte Wert einfach überschrieben wird. Eine Leerstelle besitzt den Code $20. Sie müssen also die Zahl 44 in der dritten Zeile durch 20 ersetzen. Nach Drücken der Taste <RETURN> werden die Werte in die Speicherzellen übertragen. Die ASCII-Darstellung wird ebenfalls auf den richtigen Stand gebracht. Das "D" verschwindet und wird durch eine Leerstelle ersetzt. Die Änderung können Sie jetzt leicht prüfen. Verlassen Sie den Monitor mit dem Kommando "X", drücken Sie dann die Taste <F2>, oder geben Sie mit dem BASIC-Befehl KEY die gesamte Belegung der Funktionstasten aus.

Rufen Sie jetzt den Monitor wieder auf, und stellen Sie den Speicherbereich von $1000 bis $1017 auf dem Bildschirm dar. Bei der Speicheränderung von oben haben Sie überhaupt kein Kommandozeichen eingegeben, aber trotzdem hat das Monitorprogramm erkannt, daß Sie den Speicher ändern wollen. Die Lösung des Rätsels steckt im Größerzeichen, das am Zeilenanfang ausgegeben wird. Dies ist das Befehlszeichen zum Ändern des Speichers. Es kann natürlich auch einzeln eingegeben werden. Die Speicherzelle, an der das Zeichen "D" vor der Änderung abgelegt war, hat die Adresse $1011, und das Zeichen "D" selbst besaß den Code $44. Um die Änderung rückgängig zu machen, genügt also folgende Befehlszeile:

```
>1011 44
```

Dem Kommandozeichen ">" folgt die Adresse der Speicherzelle und darauf deren neuer Wert. Wenn Sie z.B. auch noch den Inhalt der Zelle $1012, also der nachfolgenden Adresse, ändern möchten, können Sie den Wert direkt folgen lassen. Dies könnte wie folgt aussehen:

```
>1011 44 4C
```

Bei der Eingabe haben Sie gemerkt, daß der Monitor die Befehlszeile sofort auf 8 bzw. 16 Ausgabewerte ergänzt und auch die ASCII-Darstellung hinzufügt.

Wird versucht, die Speicherinhalte des ROMs zu ändern, was vollkommen unmöglich ist, wird ein Fehler in der Form eines Fragezeichens angezeigt.

Neben Befehlen zum Anzeigen und Ändern einzelner Speicherinhalte kann auch ein ganzer Speicherbereich auf einmal bearbeitet werden. Mit dem Befehl "T" wird ein Speicherbereich kopiert. Um die gesamte Funktionstastenbelegung in den freien Speicherbereich ab der Adresse $1300 zu kopieren, wird folgender Befehl verwendet:

```
T 1000 10FF 1300
```

Der erste Parameter ist die Anfangsadresse und der zweite die Endadresse des gewünschten Speicherbereichs, der als Quelle dient. Der dritte und letzte Zahlenwert ist die Startadresse des Zielbereichs. Geben Sie den Befehl ein, und stellen Sie dann mit

```
M 1300 1310
```

den Anfang der Kopie dar. Sie sehen wieder die Belegungstexte der Funktionstasten. Quell- und Zielbereich dürfen beliebig im Speicher liegen und beliebig groß sein. Auch hier wird wieder, ähnlich wie beim Ändern des Speichers mit dem Befehl ">", geprüft, ob in eine ROM-Speicherzelle geschrieben wurde, um dann eventuell einen Fehler anzuzeigen.

Ein weiterer Befehl durchsucht den Speicherbereich nach einer beliebigen Folge von Werten. Um z.B. innerhalb der Funktionstastenbelegung alle Texte zu finden, die mit einem Wagenrücklauf (CR=13) abgeschlossen wurden, werden im Speicherbereich einfach alle Speicherzellen gesucht, die den Wert $0D enthalten:

```
H 1000 10FF 0D
01020  01027  01031  01031  01036
0103E  01043  01047  0104C
```

Der erste Parameter enthält wieder die Anfangsadresse und der zweite Parameter die Endadresse des Speicherbereichs. Diesem folgen beliebig viele Bytewerte, die das Muster für die Suche bilden. Als Ausgabe stellt der Befehl die Adressen der Speicherzellen dar, die den gesuchten Wert enthalten. Hier ein weiteres Beispiel:

```
H 1300 13FF 41 44
1014
```

Das gesuchte Muster besteht jetzt aus zwei Bytes. Die ausgegebene Adresse zeigt auf das erste Byte des Paars.

Eine spezielle Eingabeform erleichtert die Suche nach Textstücken. Ein Muster, eingeleitet durch einen Apostroph, wird als Text interpretiert:

```
H 1300 13FF 'HELP
1048
```

Diese Befehlszeile sucht in den Funktionstexten das Wort "HELP". Dieser Text ist die Belegung der HELP-Taste. Auf normalem Weg ist dieser Belegungstext nicht zu ändern. Geben Sie jetzt aber einmal mit

```
M 1048 104B
```

den gefundenen Speicherbereich aus. Jetzt z.B. die Tastenbelegung mit dem Text "LIST" zu belegen, ist eine Leichtigkeit. Führen Sie dies einmal durch. Auf dem Bildschirm müßte folgende Zeile stehen:

```
>01048 4C 49 53 54 0D 00 FF 00:LIST.._.
```

Verlassen Sie den Monitor, und überprüfen Sie durch Drücken der Taste <HELP> die Änderung.

Für den nächsten neuen Befehl benötigen wir die Kopie im Speicherbereich ab der Adresse $1300. Es lassen sich mit dem Kommando "C" beliebige Teile des Speichers miteinander vergleichen. Auf dem Bildschirm werden dann die Adressen ausgegeben, deren Inhalte nicht identisch sind. Folgende Befehlseingabe zeigt die Änderung an der Belegung der <HELP>-Taste:

```
   C   1300    13FF   1000
1048   1049    104A   104B
```

Der Befehl prüft, ob der Speicher ab der Adresse $1000 identisch mit dem Bereich $1300–$13FF ist. Die ausgedruckten Adressen entsprechen der Position des Belegungstextes der <HELP>-Taste. Die Änderung läßt sich ganz leicht mit dem Kopierbefehl "T" aufheben. Es wird die Kopie ab der Adresse $1300 dazu zur Adresse $1000 zurückkopiert:

```
T 1300 13FF 1000
```

Das Ergebnis können Sie z.B. durch einen HEX-Dump des Speichers überprüfen. Mit einem speziellen Befehl wird ein Speicherbereich einheitlich mit einem Wert gefüllt. Z.B. füllt folgendes Kommando

```
F 1300 13FF EA
```

den Speicher von der Adresse $1300 bis $13FF mit dem Wert $A.

Programmierung in Maschinensprache mit dem Monitor

Für die ersten Versuche soll mit dem Assembler EDASS ein Programm erstellt werden. An diesem können einige Befehle ausprobiert werden.

```
        .BANK  15
        *=     $1300
        .OBJ   M
ZAHL1   =      250
ZAHL2   =      251
ERG     =      253
        LDA    ZAHL1        ;DEN AKKUMULATOR MIT DER 1. ZAHL LADEN
        CLC
        ADC    ZAHL2        ;DIE ZWEITE ZAHL ADDIEREN
        STA    ERG          ;DAS ERGEBNIS SPEICHERN
        RTS                 ;DAS PROGRAMM VERLASSEN
        .END
```

Geben Sie dieses kurze Programm ein, assemblieren Sie es, und rufen Sie wieder den Monitor auf.

Das Maschinenprogramm wurde ab der Adresse $1300 abgelegt und endet, wenn man nachrechnet, bei der Adresse $1307. Folgender Befehl macht den Speicherbereich z.B. sichtbar:

```
M 1300 1307
>01300 A5 FA 18 65 FB 85 FD 60:__.__.__
```

Ein Rückschluß von diesen Zahlen auf den Programmtext ist sehr schwer.

Mit einem speziellen Programm ließe sich der Assembliervorgang jedoch umkehren. Aus den Zahlenwerten, die die Prozessorbefehle darstellen, könnten wieder Mnemoniks und Operandendarstellungen werden. Ein derartiges Programm wird Disassembler genannt. Es erzeugt aus jedem beliebigen Maschinenprogramm, das in den Speicher geschrieben wird, die Mnemonik-Darstellung. Im Monitor wird der Disassembler mit dem Kommando "D" aufgerufen. Hier die nötige Befehlszeile, um das obige Beispielprogramm darzustellen:

```
D 1300 1307
. 01300  A5 FA    LDA $FA
. 01302  18       CLC
. 01303  65 FB    ADC $FB
. 01305  85 FC    STA $FD
. 01307  60       RTS
```

Wie Sie am Ergebnis sehen, nimmt der Disassembler wirklich nur die im angegebenen Speicherbereich liegenden Werte als Grundlage. Er kennt keine

Hilfsmittel wie Labels. Deshalb erscheint in der Ausgabe auch nicht LDA ZAHL1, sondern LDA $FA, die rückübersetzte Form des Prozessorbefehls. Sie merken schon, daß der Disassembler zwar ein sehr einfaches, aber wirkungsvolles Hilfsmittel ist.

In früheren Kapiteln haben Sie erfahren, daß ein Maschinenbefehl aus ein bis drei Bytes besteht. Das erste Byte enthält grundsätzlich die Information, was der Prozessor bei diesem Befehl tun soll und wie er die benötigten Daten erhält. Erst in den nachfolgenden Bytes erfolgen die Angaben über Speicheradressen.

Ein Byte kann 256 verschiedene Werte aufnehmen. Der 8502 kennt aber nur ca. 150 verschiedene Befehle. Es gibt Zahlenwerte, die er nicht als Befehl interpretieren kann. Die Reaktionen darauf sind sehr unterschiedlich. Natürlich kann auch der Disassembler auf derartige Zahlenwerte stoßen. An diesen Stellen gibt er als Mnemonik drei Fragezeichen aus ("???"). Hier ein Beispiel:

```
D 1001 100A
. 1001  06 0A    ASL  $0A
. 1003  07       ???
. 1004  06 04    ASL  $04
. 1006  05 08    ORA  $08
. 1008  09 05    ORA  #$05
. 100A  47       ???
```

In diesem Beispiel erkennt der Disassembler den Inhalt der Speicherzelle $1001 als einen Maschinenbefehl. Bei der Adresse $1003 hingegen handelt es sich um keine Maschinenanweisung, folglich erscheinen in der Ausgabe drei Fragezeichen. Die nächsten sechs Bytes können wieder als Befehl interpretiert werden. Erst bei der Adresse $100A stößt der Assembler wieder auf ein Byte, das sich nicht disassemblieren läßt.

Bei Ausgaben, in denen als Mnemonik die drei Fragezeichen erscheinen, handelt es sich meist nicht um ein Maschinenprogramm, sondern um eine Datentabelle. Im Beispiel ist diese die Funktionstastenbelegung. Jeder Speicherbereich kann als Maschinenprogramm interpretiert werden, denn für den Prozessor besteht das Programm ja nur aus Zahlenwerten.

Zum Abschluß noch ein kleines Beispiel, in dem ein Stück aus dem BASIC-ROM disassembliert wird.

```
D F4000 F400F
. F4000  4C 23 40 JMP $4023
. F4003  4C 09 40 JMP $4009
. F4006  4C 4D A8 JMP $A84D
. F4009  20 CC FF JSR $FFCC
. F400C  20 7A 41 JSR $417A
. F400F  20 8D 41 JSR $418D
```

In der Befehlszeile wird mit der vordersten Ziffer die ROM-Speicherbank als Quelle für die Disassemblierung gewählt.

Bis jetzt haben Sie ein Maschinenprogramm nur angeschaut, es kann aber auch verändert werden. Der Monitor stellt hierzu einen einfachen Assembler zur Verfügung. Dieser kennt genauso viel Komfort wie der Disassembler. Ein einzelner Befehl wird in der vom Disassembler vorgeführten Form eingegeben und sofort assembliert. Hier ein Beispiel:

```
A 1300 LDA $FA
```

Nach Eingabe dieser Zeile führt das Monitorprogramm die Assemblierung durch und gibt die Zeile zusammen mit den übersetzten Zahlenwerten neu aus. Zusätzlich wird in die nächste Zeile der Kommandobuchstabe "A" und eine neue Startadresse gedruckt. Für ein komplettes Assemblerkommando muß nur noch der Maschinenbefehl eingetippt werden. Ingesamt zeigt sich folgendes Bild:

```
A 1300  A5 FA    LDA $FA
A 1302
```

Noch ein weiterer Weg, den Assembler zu aktivieren, wird vom Monitor eröffnet. Bei den Ausgaben des Disassemblers wird an jeden Zeilenanfang ein De- zimalpunkt geschrieben. Dieser ist auch ein Kommando. Es ist identisch mit der Assembleranweisung "A". Das Programm kann direkt aus dem Disassemblerlisting verändert werden. Der Cursor wird auf das Mnemonik bzw. den Adreßteil bewegt, womit sich dieser überschreiben läßt. Die Ausgabe des Maschinencodes zwischen Speicheradresse und Mnemonik wird vom Monitorprogramm übergangen. Das Beispielprogramm soll so geändert werden, daß das Ergebnis nicht in die Speicherzelle $FD (253), sondern in $FC (252) geschrieben wird:

```
D 1300 1307
. 01300  A5 FA    LDA $FA
. 01302  18       CLC
. 01303  65 FB    ADC $FB
. 01305  85 FC    STA $FD
. 01307  60       RTS
```

Den Cursor zu "$FD" bewegen, korrigieren, und nach Drücken der Taste <RETURN> zeigt sich folgendes Bild:

```
A 01305  85 FC    STA $FC
A 01307  60       RTS
```

Da Sie mit dem Ändern der Zeile den Assembler aufgerufen haben, wird gleich in der nächsten Zeile ein "A" und die nächste Startadresse ausgegeben. Auch wartet der Cursor hinter der Adresse auf Ihre Eingaben.

Bevor weitere Eigenschaften des Monitorassemblers besprochen werden, soll erklärt werden, wie ein Maschinenprogramm im Monitor gestartet wird. Folgender Befehl startet z.B. das Beispielprogramm:

```
J 1300
```

Das Programm addiert die Inhalte der Speicherzellen $FA (250) und $FB (251) und legt das Ergebnis in der Speicherzelle $FC (252) ab. Am Anfang dieses Kapitels haben Sie die nötigen Befehle kennengelernt, um Werte in Speicherzellen abzulegen und wieder auszulesen. Diese helfen Ihnen jetzt, die beiden Ausgangswerte der Addition in den Speicher zu schreiben und das Ergebnis wieder auszulesen:

```
>00FA 05 2A

J 1300

M 00FA 00FC
>000FA 05 2A 2F ...
```

In der Ausgabezeile des "M"-Befehls stellt die dritte Ausgabeposition den Inhalt der Speicherzelle $FC dar. Diese enthält das korrekte Ergebnis $05 + $2A = $2F.

Mit dem Monitor-Assembler soll jetzt ein Programm eingegeben werden, das das Zweierkomplement des Akkumulatorinhalts bildet. Wie der Eingangswert soll auch der Endwert im Akkumulator gespeichert werden. Hier das Programm, links, was Sie eingeben müssen, und rechts, was am Schluß auf dem Bildschirm zu sehen ist:

```
A 1300 EOR #$FF              A 01300  49 FF     EOR #$FF
CLC                          A 01302  38        CLC
ADC #$1                      A 01303  E9 01     ADC #$01
RTS                          A 01305  60        RTS
```

Das Programm selbst kann wieder mit

```
J 1300
```

gestartet werden. Wie werden diesem aber jetzt Parameter übergeben? Wie liest man das Ergebnis, das ja im Akkumulator übergeben wird? Beim Aufruf des Programms mit dem "J"-Befehl werden vorgegebene Werte an das Maschinenprogramm übergeben. Welche Werte voreingestellt sind, zeigt der "R"-Befehl:

```
R
    PC  SR AC XR YR SP
; 01306 00 00 00 00 F8
```

Die einzelnen Werte haben folgende Bedeutung:

PC	=	program counter	=	Programmzähler
SR	=	statusregister	=	Statusregister
AC	=	accumulator	=	Akkumulator
XR	=	X-register	=	X-Register
YR	=	Y-register	=	Y-Register
SP	=	stack-pointer	=	Stapelzeiger

Wahrscheinlich haben Sie sich schon gedacht, daß der Strichpunkt (";") am Zeilenanfang der Ausgabe das Kommando zum Ändern der gespeicherten Werte ist. Der Cursor wird einfach auf die Ausgabe positioniert, die Werte werden geändert und mit Drücken der RETURN-Taste gespeichert. Beim Starten des nächsten Maschinenprogramms werden die neu eingegebenen Werte an die einzelnen Register übergeben. Nach Abschluß des Programms werden die Registerinhalte in Speicherzellen festgehalten und können mit "R" sichtbar gemacht werden.

Dem Beispielprogramm können über den beschriebenen Weg Parameter das übergeben und von dort wieder empfangen werden:

```
R
    PC  SR AC XR YR SP
; 01306 00 3B 00 00 F8

J 1300

R
    PC  SR AC XR YR SP
; 01306 00 C5 00 00 F8
```

In einem weiteren Beispielprogramm sollen so viele Sternchen ausgegeben werden, wie der Inhalt des X-Registers anzeigt. Unten sehen Sie wieder links die Eingaben, die Sie tätigen müssen, und rechts, was auf dem Bildschirm erscheint:

```
A 1300 LDA #$2A                A 01300  A5 2A     LDA #$2A
JSR $FFD2                      A 01302  20 D2 FF  JSR $FFD2
DEX                            A 01305  CA        DEX
BNE $1302                      A 01306  D0 FA     BNE $1302
RTS                            A 01308  60        RTS
```

Am abgedruckten Text sehen Sie, daß bei Branch-Befehlen direkt die Zieladresse eingegeben werden kann. Die Umrechnung übernimmt die Assemblerroutine. Das Programm setzt jetzt eine ROM-Routine ein, entsprechend muß auch beim Aufruf mit dem "J"-Befehl die ROM-Bank in der Startadresse gewählt werden. Auch wenn die ROM-Bank gewählt wird, liegt innerhalb des

Adreßbereichs $0000 bis $3FFF noch die RAM-Bank 0. In diesem Adreßraum sind keine ROMs installiert. Hier der komplette Programmablauf, mit Übergabe der Parameter und Ausgabe der Sternchen:

```
R
    PC  SR AC XR YR SP
; 01306 00 C5 0A 00 F8

J F130**********
```

Testen Sie die Wirkung des Programms, indem Sie den vorgegebenen Inhalt des X-Registers ändern.

Durch die internen Abläufe im Monitorprogramm werden die Sterne hinter dem Befehl und nicht in einer neuen Zeile ausgegeben. Dieser Umstand läßt sich leicht durch Ausgabe eines Wagenrücklaufs vor Ausführung der Schleife beheben. Zwei Befehle sind hierfür erforderlich, aber diese müßten vor dem "LDA $2A" eingefügt werden. Dazu müßte das gesamte bisherige Programm im Speicher verschoben werden bzw. neu eingegeben werden. Sie sehen also, welche Vorteile ein Assembler wie EDASS gegenüber einem so einfachen wie dem des Monitors hat.

Mit noch einem zweiten Befehl wird ein Maschinenprogramm vom Monitor aufgerufen. Das Kommando "G" startet das Programm, erwartet jedoch zur Rückkehr in den Monitor einen BRK-Befehl. Wenn Sie sich erinnern, wurde bisher zum Rücksprung immer ein RTS-Befehl herangezogen. In vielen anderen Monitorprogrammen findet man nur diesen Befehl zum Programmaufruf. Durch folgende Befehlszeile wird das letzte Beispiel entsprechend geändert:

```
A 1308 BRK
```

Jetzt darf das Programm mit

```
R    PC  SR AC XR YR SP
;  01306 00 C5 0C 00 F8
G  F1300
```

gestartet werden. Wird diese Forderung des "G"-Befehls nicht beachtet, kommt es oft zum Absturz des Computers.

Der BRK-Befehl am Ende des Programms ist noch in einer weiteren Hinsicht hilfreich. Der dabei ausgelöste softwaremäßige Interrupt ruft wieder den Monitor auf, und bei jedem Monitoraufruf werden die aktuellen Werte der Register dargestellt. Jetzt werden Ihnen auch die Ausgaben beim Starten des Monitors klar. Die beschriebene Eigenschaft des BRK-Befehls wird oft gezielt zur Fehlersuche verwendet. Nehmen wir einmal an, im Beispielprogramm hat

sich im Schleifenrumpf ein Fehler ergeben, dessen Ursache aber noch nicht ganz geklärt ist. Es wird jetzt einfach der Branch-Befehl, der die Schleife schließt, durch einen BRK-Befehl ersetzt:

```
A 1306 BRK
```

Beim Starten des Programms wird jetzt der Schleifenrumpf durchlaufen und dann mit BRK wieder der Monitor aufgerufen, wobei die Registerinhalte dargestellt werden. An Hand dieser können z.B. Rückschlüsse auf einen Fehler gewonnen werden. Bei größeren Programmen ist jetzt auch der Zugang zu diversen Speicherzellen frei. Diese enthalten Werte des Programmablaufs. Eine genaue Untersuchung bringt oft neue Erkenntnisse.

Diskettenbedienung durch den Monitor

Der Monitor kennt insgesamt vier Kommandos, die sich auf Operationen mit einem Disketten- bzw. Kassettengerät beziehen. Das Kommando "S" speichert einen beliebigen Speicherbereich aus einer beliebigen Bank auf ein Ausgabegerät. Die Befehlsform sieht wie folgt aus:

```
S "TEST",08,1300,1307
S "ROM",01,F4100,F4150
```

Im ersten Beispiel wird der Speicherbereich $1300–$1307 auf das Diskettengerät mit der Geräteadresse 8 gespeichert, und im zweiten Beispiel wird ein Stück des BASIC-ROMs auf Kassette gesichert. Im Endprodukt ist der Befehl mit der BASIC-Anweisung BSAVE zu vergleichen.

Das Gegenstück zum Speicherbefehl bildet das "L"-Kommando. Das zu ladende Speicherstück bzw. Programm kann natürlich auch von einem anderen Programm als dem Monitor erzeugt worden sein, so z.B. von EDASS. In der Befehlsanwendung gibt es zwei verschiedene Formen:

```
L "TEST",08

L "ROM",08,1300
```

Bei der ersten Befehlsform wird das Programm ab der in der Datei gespeicherten Adresse in den Speicher geladen. In der zweiten Anwendungsform wird das Programm ab der im Befehl gewählten Adresse geladen. Der Befehl ist wieder mit der BASIC-Anweisung BLOAD zu vergleichen.

Wenn ein "Save"- und ein "Load"-Kommando vorhanden sind, darf eine entsprechende "Verify"-Anweisung nicht fehlen. Die Eingabeform des Befehls entspricht der des "L"-Kommandos. Hier zwei Beispiele:

```
V "TEST",08,1300

V "ROM",0A
```

Der letzte Befehl, ein Klammeraffe (@), bezieht sich nur auf die Diskettenstation. Zusammen mit einer gleichzeitig eingegebenen Geräteadresse druckt er den Diskettenstatus aus. Außerdem darf ein Diskettenkommando eingetippt werden, das dann an die entsprechende Floppy-Disk-Station gesandt wird. Wieder ein paar Beispiele:

```
@08
00, OK, 00,00

@0A,S0:TEST
01, FILES SCRATCHED,01,00

@08,I0
00, OK, 00 00
```

Zum Abschluß dieses Kapitels eine Übersicht über alle Befehle des Monitors:

A (assemble)	assembliert einen 6502/8502-Assemblerbefehl.
C (compare)	vergleicht zwei Speicherbereiche byteweise.
D (disassemble)	disassembliert einen Speicherbereich.
F (fill)	füllt einen Speicherbereich mit einem Wert.
G (go)	startet ein Maschinenprogramm (Ende mit BRK).
H (hunt)	sucht in einem Speicherbereich ein Muster.
J (jump)	startet ein Maschinenprogramm (Ende mit RTS).
L (load)	lädt ein Programm von Diskette oder Kassette.
M (memory)	zeigt den Inhalt eines Speicherbereichs an.
R (register)	zeigt die Startwerte der Prozessorregister an.
S (save)	sichert einen Speicherbereich.
T (transfer)	kopiert einen Speicherbereich.
V (verify)	vergleicht den Speicher mit einer Programmdatei.
X (exit)	beendet das Monitorprogramm.
>	modifiziert den Inhalt von Speicherzellen.
.	ist identisch mit dem "A"-Kommando.
;	ändert die Startwerte der Prozessorregister.
@	zeigt Disk-Status oder sendet einen Diskbefehl.

Kapitel 12

Weitere Funktionen von EDASS

Bedingte Assemblierung

Bei bedingter Assemblierung werden Teile des Assemblerprogramms nur bei Eintritt einer bestimmten Bedingung assembliert. Welchen Sinn hat es überhaupt, Teile nicht zu assemblieren? Mit der bedingten Assemblierung können z.B. Fehlerdiagnose-Routinen auf Wunsch in das Programm eingebaut werden.

Bei EDASS wird die bedingte Assemblierung durch die Pseudo-Befehle .IFNE (identisch mit .IF), .IFEQ, .IFPL und .IFMI eingeleitet und mit .ENDIF beendet. Alle dazwischenliegenden Befehle werden nur bei Eintritt der speziellen Bedingung assembliert. Diese Bedingung besteht aus einem mathematischen Ausdruck, der nach den einleitenden Befehlen .IFNE,... genannt wird.

Jedes der vier Befehlswörter assembliert den eingeschlossenen Programmteil bei einem anderen Ergebnis, der Befehl .IFNE (NE=Not Equal to zero), wenn der Ausdruck einen Wert verschieden von null hat, der Befehl .IFEQ (EQ=EQual to zero), wenn der Ausdruck den Wert Null hat, der Befehl .IFPL (PL=PLus), wenn der Ausdruck einen positiven Wert hat und schließlich der Befehl .IFMI (MI=MInus), wenn der Ausdruck einen negativen Wert hat. Dazu ein Beispielprogramm:

```
VERSION  =       0
BSOUT    =       $FFD2
         .IFEQ   VERSION          ;BEI VERSION=0 ASSEMBLIEREN
         LDA     #"A"
         .ENDIF
         .IFNE   VERSION          ;BEI VERSION<>0 ASSEMBLIEREN
         LDA     #"X"
         .ENDIF
         JSR     BSOUT            ;DAS VERSIONSZEICHEN AUSGEBEN
         LDA     #13              ;DEN AKKU. MIT CODE VON CR LADEN
         JMP     BSOUT            ;DAS ZEICHEN AUSGEBEN (+ENDE)
```

Das Label VERSION entscheidet darüber, welche Version des Programms assembliert wird. Hat das Label den Wert Null, ist die Bedingung beim Befehl .IFEQ erfüllt, und es wird der Befehl LDA #"A" übersetzt. Bei einem Wert ungleich null ist nur die Bedingung beim Befehl .IFNE erfüllt, und entspre-

chend wird nur der Befehl LDA #"X" assembliert. Tippen Sie das Programm doch einfach ein,und probieren Sie es aus. Durch geschickte Formulierung der Bedingungen lassen sich z.B. auch drei und mehr verschiedene Versionen gleichzeitig programmieren. Hier wieder das entsprechende Beispiel:

```
VERSION   =       3
BSOUT     =       $FFD2
          .IFEQ   VERSION-1    ;BEI VERSION=1 ASSEMBLIEREN
          LDA     #"A"
          .ENDIF
          .IFEQ   VERSION-2    ;BEI VERSION=2 ASSEMBLIEREN
          LDA     #"X"
          .ENDIF
          .IFEQ   VERSION-3    ;BEI VERSION=3 ASSEMBLIEREN
          LDA     #"*"
          .ENDIF
          JSR     BSOUT        ;DAS VERSIONSZEICHEN AUSGEBEN
          LDA     #13          ;DEN AKKU. MIT CODE VON CR LADEN
          JMP     BSOUT        ;DAS ZEICHEN AUSGEBEN (+ENDE)
```

Bei den drei Abfragen ist jeweils die Bedingung nur erfüllt, wenn die Differenz aus der Versionsnummer und dem Inhalt des Labels VERSION null ergibt. Sie erkennen daran auch, daß sich die Reihe der Versionen beliebig fortsetzen ließe. Mit den weiteren Bedingungsbefehlen könnten, z.B. ab Version 2 (.IFPL VERSION-2), weitere spezielle Routinen assembliert werden.

Bei diesem einfachen Beispiel ist der Einsatz von bedingter Assemblierung sicher noch nicht sinnvoll. Aber EDASS z.B. liegt auf der Diskette in zwei Versionen vor. Beide Versionen sind in einem Programm enthalten. Durch den Wert eines Labels wird an verschiedenen Stellen des Programms beim Assemblieren entschieden, welche Version zu erzeugen ist. Ohne bedingte Assemblierung müßte das Programm ein zweites Mal programmiert werden.

Makroassemblierung

Häufig wird dieselbe Befehlssequenz in Programmen mehrmals verwendet und bestimmte Befehlssequenzen finden Sie in ähnlicher Form in jedem Programm. Betrachten Sie nur die Beispiele aus all den vorangegangenen Kapiteln. Wie oft wurde dort eine kleine Schleife mit dem X-Register aufgebaut, eine 16-Bit-Zahl addiert oder eine 16-Bit-Zahl nach links verschoben. Bei großen Programmen häuft sich viel unnötige Tipparbeit an. Um diese zu umgehen, hat man sogenannte Makros eingeführt.

Makros sind eine Gruppe von Befehlen, die eine Programmsequenz bilden. Ihnen wird ein Name zugewiesen, unter dem sie zum Gebrauch angesprochen werden. Wird jetzt in einem Programm genau diese Befehlsgruppe eines be-

stimmten Makros benötigt, genügt es, im Programmtext den Namen des Makros zu nennen. Der Assembler setzt an dieser Stelle die Befehle des Makros ein. Natürlich kann jetzt an einer zweiten Stelle das Makro wieder aufgerufen werden. Auch dort setzt der Assembler wieder die Befehle des Makros in das Programm ein.

Es ist wichtig, daß Makros nicht mit Unterprogrammen verwechselt werden. Unterprogramme existieren im gesamten Maschinenprogramm nur ein einziges Mal. Der Unterprogrammaufruf wird durch einen Prozessorbefehl eingeleitet, womit dieser seine Arbeit im Unterprogramm fortsetzt. Das Makro wird bei jedem Aufruf neu assembliert.

Wird es z.B. viermal im Programm verwendet, so ist es auch viermal in voller Länge im Maschinenprogramm enthalten. Für den Prozessor ist es ein fortlaufendes Programm. Mit diesen Erklärungen wird Ihnen auch der Nachteil der Makros gegenüber den Unterprogrammen deutlich. Das Makro verbraucht bei jeder Verwendung Speicherplatz, das Unterprogramm nur einmal.

Makros sind sinnvoll, wenn Tipparbeit gespart werden soll, wenn es auf Geschwindigkeit ankommt (ein Unterprogrammaufruf benötigt Zeit) oder wenn die entsprechende Konstruktion nicht durch ein Unterprogramm ersetzt werden kann.

Jetzt zu praktischeren Dingen: Mit welchen Befehlen wird bei EDASS Makroassemblierung ermöglicht? Vor der Verwendung müssen ein Makro bzw. die Befehle, die ein Makro bilden, definiert werden. Der Pseudo-Befehl .MACRO leitet diese Definition ein, und der Befehl .ENDM beendet sie wieder. Alle zwischen beiden Befehlen eingeschlossenen Assemblerbefehle bilden das Makro. Der Name des Makros wird wie ein Label vor den Befehl .MACRO geschrieben. Hier ein Beispiel für eine Definiton:

```
XNACHY   .MACRO          ;START DER DEFINITION
         TXA             ;DIE MAKRO-BEFEHLE
         TAY
         .ENDM           ;ENDE DER DEFINITION
```

Erreicht der Assembler diese Zeilen, merkt er sich, daß die Befehlssequenz TXA, TAY als Makro mit dem Namen XNACHY definiert wurde.

Aufgerufen wird das Makro jetzt mit:

```
/XNACHY
```

Der Schrägstrich sagt dem Assembler, daß es sich bei dem nachfolgenden Namen um einen Makronamen handelt. Es werden jetzt die Befehle des Makros, hier TXA und TAY, assembliert und in das Programm eingefügt.

Ein komplettes Programm könnte wie folgt aussehen:

```
        .BANK   15
        *=      $1300
        .OBJ    M
;
XNACHY  .MACRO                  ;START DER DEFINITION
        TXA                     ;DIE MAKRO-BEFEHLE
        TAY
        .ENDM                   ;ENDE DER DEFINITION
;
        LDX     #100            ;X-REG. MIT DEM WERT 100 LADEN
        /XNACHY                 ;INHALT DES X-REG. NACH Y-REG.
        STY     250             ;Y-REG. IN 250 SPEICHERN
        RTS                     ;DAS PROGRAMM VERLASSEN
        .END
```

Wichtig ist, daß das Makro vor der ersten Verwendung definiert wird. Denn wie soll der Assembler eine Befehlsfolge einsetzen, wenn er diese noch gar nicht kennt? Bei diesem Beispiel ist der Einsatz eines Makros noch nicht sinnvoll. Man vergleiche das Programm nur mit einer ohne ein Makro programmierten Lösung:

```
        .BANK   15
        *=      $1300
        .OBJ    M
        LDX     #100            ;X-REG. MIT DEM WERT 100 LADEN
        TXA                     ;INHALT DES X-REG. IN AKKU.
        TAY                     ;INHALT DES AKKU. IN DAS Y-REG.
        STY     250             ;Y-REG. IN 250 SPEICHERN
        RTS                     ;DAS PROGRAMM VERLASSEN
        .END
```

Bei längeren Programmen kann diese Befehlsfolge z.B. häufig Gebrauch finden. Der Einsatz eines Makros gewinnt an Bedeutung.

Die Makros bei EDASS bieten eine weitere Einrichtung. Der Befehlssequenz können Parameter übergeben werden. Dies sieht dann wie folgt aus:

```
ADD     .MACRO  (SUM1,SUM2)
        LDA     SUM1
        CLC
        ADC     SUM2
        STA     SUM2
        .ENDM
;
        /ADD    (250,252)       ;SPEICHERZELLE 250 PLUS 252
        /ADD    (251,253)       ;SPEICHERZELLE 251 PLUS 253
        RTS                     ;DAS PROGRAMM VERLASSEN
```

Bei der Definition werden nach dem Befehlswort .MACRO, in Klammern eingeschlossen, die Namen der Parameter aufgelistet. Diese Namen sind nichts

anderes als Labels, denen bei Aufruf des Makros ein Wert zugewiesen wird. Die Befehle des Makros greifen auf diese Labels zurück. Beim Makroaufruf werden entsprechend nach dem Makronamen Werte, in Klammern eingeschlossen, genannt. Genau diese Werte werden den Labels zugewiesen und bei der Assemblierung des Makros verarbeitet.

Das Makro wird flexibel und kann an verschiedene Programmpositionen angepaßt werden. Im Beispiel werden auf diese Weise einmal die Inhalte der Speicherzellen 250 und 252 addiert und das andere Mal die Inhalte der Zellen 251 und 253.

Noch ein weiteres Beispiel für Makros. Diesmal wird in einem Makro die Multiplikation mit 10 definiert.

```
SUM1    =   250
SUM2    =   251
;
MUL10   .MACRO (ZAHL)
        LDA     ZAHL            ;DEN ZAHLENWERT LADEN
        ASL     A               ;NACH LINKS SCHIEBEN (*2)
        ASL     A               ;NACH LINKS SCHIEBEN (*4)
        CLC                     ;DEN UEBERTRAG LOESCHEN
        ADC     ZAHL            ;DEN ZAHLENWERT ADDIEREN (*5)
        ASL     A               ;NACH LINKS SCHIEBEN (*10)
        STA     ZAHL            ;DAS ERGEBNIS SPEICHERN
        .ENDM
;
        /MUL10 (SUM1)           ;SUM1 * 10
        /MUL10 (SUM2)           ;SUM2 * 10
        LDA     SUM1            ;DIE ERGEBNISSE ADDIEREN
        CLC
        ADC     SUM2
        STA     SUM1
        RTS                     ;DAS PROGRAMM VERLASSEN
```

Das Programm multipliziert jeweils die Inhalte der Speicherzellen 250 und 251 mit 10 und addiert diese Teilergebnisse. Das Endergebnis wird in die Speicherzelle 250 gespeichert.

Das Problem ließe sich genausogut mit einem Unterprogramm lösen. Hier würde das Unterprogramm den zu multiplizierenden Wert im Akkumulator übernehmen und das Ergebnis wieder im Akkumulator zurückgeben.

Es wird jedoch eine zusätzliche Speicherzelle benötigt, um den Eingangswert für die Multiplikation zwischenzuspeichern.

Eine Lösung sähe wie folgt aus:

```
SUM1      =       250
SUM2      =       251
ZWISCHEN  =       252
          LDA     SUM1        ;DEN 1. WERT LADEN
          JSR     MUL10       ;UND *10
          STA     SUM1        ;UND SPEICHERN
          LDA     SUM2        ;DEN 2. WERT LADEN
          JSR     MUL10       ;UND *10
          STA     SUM2        ;UND SPEICHERN
;
          LDA     SUM1        ;DIE ERGEBNISSE ADDIEREN
          CLC
          ADC     SUM2
          STA     SUM1
          RTS                 ;DAS PROGRAMM VERLASSEN
;
MUL10     STA     ZWISCHEN    ;DEN WERT MERKEN
          ASL     A           ;NACH LINKS SCHIEBEN (* 2)
          ASL     A           ;NACH LINKS SCHIEBEN (* 4)
          CLC                 ;DEN UEBERTRAG LOESCHEN
          ADC     ZWISCHEN    ;DEN ZAHLENWERT ADDIEREN (* 5)
          ASL     A           ;NACH LINKS SCHIEBEN (* 10)
          RTS                 ;DAS UNTERPROGRAMM VERLASSEN
```

Diese Lösung ist nur wenig kürzer als bei der Verwendung von Makros.

Sie haben jetzt ein Beispiel, an dem Sie beide Methoden vergleichen können. Im praktischen Einsatz sollten Sie zuerst versuchen, das Problem mit einem Unterprogramm zu lösen und erst dann Makros einsetzen. Sie können auch in Programmen häufig wiederkehrende Befehlsfolgen in Makros fassen und diese in einer Datei sammeln.

Ihr eigentliches Programm könnte dann im wesentlichen nur aus Makroaufrufen bestehen. Zwei Beispiele für derartige Makrosammlungen finden Sie auf der EDASS-Diskette. Im zweiten Teil des Buches werden diese ausführlich beschrieben.

Der Reassembler

In Kapitel 11 haben Sie bereits den Disassembler des Monitors kennengelernt. Sinnvoll wäre z.B. eine Einrichtung, die das disassemblierte Maschinenprogramm nicht nur anzeigt, sondern auch wieder in ein Quellprogramm verwandelt, das mit EDASS editiert werden kann. Genau diese Aufgabe nimmt der Reassembler wahr.

Schon am Namen erkennen Sie, daß dieser Befehl nicht einfach disassembliert, sondern vielmehr den Assembliervorgang mit all seinen Vorteilen und Einrichtungen umkehrt. Aufgerufen wird der Reassembler mit dem EDASS-

Befehl !REASS, gefolgt vom Namen des neu zu erzeugenden Quellprogramms und der Anfangs- und Endadresse des zu reassemblierenden Speicherbereichs. Hier ein paar Beispiele:

```
!REASS "TEST",$1300,$1400
!REASS "ROM",$4000,$4010
!REASS "EDITOR",49152,49200
```

Anzumerken ist noch, daß der Speicherbereich in jeder Speicherbank liegen darf. Vor der Anwendung des !REASS-Befehls muß deshalb mit dem BASIC-Befehl BANK die gewünschte Bank ausgewählt werden. Sollten Sie nicht mehr genau wissen, wie die verschiedenen Speicherkombinationen ausgewählt werden, lesen Sie einfach nochmals in Kapitel 9 nach.

An einem Beispiel sollen die ersten Eigenschaften des Reassemblers demonstriert werden. Es wird hierzu ein Programmstück aus dem ROM des C128 reassembliert. Das Programm kopiert eine Programmsequenz aus dem ROM ins RAM.

```
BANK 15
!REASS "ROM",$EE9B,$EEA7
```

Mit dem Befehl BANK 15 wurden die ROMs des C128 als Speicherbank gewählt, und mit dem REASS-Befehl wurde schließlich der Speicherbereich reassembliert. Die Adreßangaben $EE9B und $EEA7 begrenzen den gewünschten Speicherbereich. Das als Endprodukt erzeugte Assemblerprogramm ist unter dem Namen "ROM" ansprechbar. Sie können daran Änderungen wie bei einem frisch eingegebenen Programm vornehmen. Rufen Sie den Editor einmal mit !EDIT "ROM" auf, und betrachten Sie das Ergebnis:

```
        .BANK  $0F
        *=     $EE9B
        LDA    $EEA0,X
        STA    $0314
LEEA1   LDA    LEEA1,X
        STA    $0315
        RTS
        .END
```

Am Programmtext fällt zuerst auf, daß Pseudo-Befehle für die Wahl der Banknummer und Startadresse in den Text aufgenommen wurden. Die Lage des Programms ist also genau festgehalten. Auch ein .END-Befehl fehlt nicht. Dieser wurde ganz korrekt ans Programmende angehängt. Jetzt aber zum eigentlichen Programmtext: Jeder Befehl wurde wie beim Disassembler übersetzt. Der zweite Lade-Befehl weist jedoch eine Besonderheit auf. Der Reassembler hat erkannt, daß sich die Operandenadresse innerhalb des Programms befindet. Wie in einem Assemblerprogramm wurde folglich für diese

Programmstelle ein Label verwendet. Den Labelnamen hat der Reassembler aus dem Buchstaben "L" und der Adresse der Programmposition gebildet und dann am Anfang der entsprechenden Zeile und im Operanden eingesetzt.

Noch ein weiteres Beispiel, das mit folgenden Befehlen erzeugt wird:

```
BANK 15
!REASS "SCHLEIFE",$E253,$E262
```

Das Ergebnis dieser beiden Befehl sieht wie folgt aus:

```
          BANK    $0F
          *=      $E253
          LDX     #$08
LE255     LDA     $E262,X
          STA     $01,X
          DEX
          BNE     LE255
          STX     $D030
          JMP     $0002
          .END
```

Auch hier hat der Reassembler wieder an geeigneter Stelle ein Label eingesetzt. Der Branch-Befehl besitzt als Operanden ein Label, mit dem auf die entsprechende Zeile verzweigt wird.

Der REASS-Befehl kann aber noch mehr als diese einfachen Labelbildungen. Um diese Funktion zu testen, wird zuerst ein kleines Programm mit dem Assembler erzeugt und direkt im Anschluß mit dem Reassembler wieder zurückverwandelt.

```
          .BANK   15
          *=      $1300
          .OBJ    M
BSOUT     =       $FFD2
          LDX     #5          ;SCHLEIFENZAEHLER MIT 5 STARTEN
SCHLEIFE  LDA     #"*"        ;AKKU. MIT CODE VON "*" LADEN
          JSR     BSOUT       ;DAS ZEICHEN AUSGEBEN
          DEX                 ;DEN SCHLEIFENZAEHLER VERMINDERN
          BNE     SCHLEIFE    ;ENDE DER SCHLEIFE ?, NEIN =>
          RTS                 ;DAS PROGRAMM VERLASSEN
          .END
```

Der Assembler wird mit !ASSEMBLER "BEISPIEL" gestartet und der Reassembler direkt danach mit

```
BANK 15
!REASS "TEST",$1300,$130A
```

Als Ergebnis erhalten Sie:

```
                .BANK   $0F
                *=      $1300
                LDX     #$05
SCHLEIFE        LDA     #$2A
                JSR     BSOUT
                DEX
                BNE     SCHLEIFE
                RTS
                .END
```

Wie im ersten Beispiel wurden auch jetzt Labels statt Adressen in den Quelltext eingefügt. Aber woher wußte der Reassembler, daß als Label zum Aufbau der Schleifenverzweigung der Name "SCHLEIFE" verwendet wurde? Nach dem Assemblieren befinden sich die Labelwerte noch im Speicher. Sie können von dort einzeln gelesen und für weitere Auswertungen genutzt werden. Genau dies macht der Reassembler. Sobald beim Reassemblieren ein Wert auftaucht, der auch als Label existiert, wird dessen Name in das Programm eingebaut.

Es ist auch möglich, nachträglich Labels zu definieren. Geben Sie folgende Befehle ein:

```
!ERASE "TEST"
!ASSEMBLER "BEISPIEL"
!LET STERN=42
```

Hiermit wird zu den Labels BSOUT und SCHLEIFE noch STERN hinzugefügt. Reassemblieren Sie jetzt das Programm nochmals mit:

```
BANK 15
!REASS "TEST",$1300,$130A
```

Es taucht jetzt, wie Sie sicher schon erwartet haben, folgende Zeile auf:

```
    .
    .
LDA       #STERN
    .
    .
```

In einem dritten Durchlauf werden vor dem Aufruf des Reassemblers alle Labels gelöscht. Geben Sie dazu folgende Befehle ein:

```
!ERASE "TEST"
!CLEAR
BANK 15
!REASS "TEST",$1300,$130A
```

Das Endprodukt dieses Durchlaufs erinnert wieder mehr an das Ergebnis eines Disassemblers:

```
            .BANK   $0F
            *=      $1300
            LDX     #$05
L1302       LDA     #$2A
            JSR     $FFD2
            DEX
            BNE     L1302
            RTS
            .END
```

Die Wirkung noch im Speicher befindlicher Labels wird beim Vergleich der ersten zwei Durchläufe mit dem letzten ganz deutlich. Welche weiteren Leistungen der REASS-Befehl bietet und wie Sie komfortabel mit Labels umgehen, erfahren Sie im zweiten Teil des Buches. Dort werden alle Befehle ausführlich beschrieben.

Kapitel 13

Starten von EDASS

Das Programm läßt sich auf verschiedene Weisen laden und starten, z.B. mit dem BASIC-Befehl

```
RUN "EDASS 128"
```

Die EDASS-Diskette wurde boot-fähig gemacht. Das Programm läßt sich deshalb mit dem Befehl

```
BOOT
```

oder durch Einlegen der Diskette beim Einschalten des Computers bzw. beim Reset starten. Dies ist sicherlich der bequemste Weg.

Nach einem Anfangsbild erscheint die Startmeldung:

```
**** EDASS - EDITOR+ASSEMBLER ****
        (C) BY FRANK MUELLER
           C128 VERSION
READY.
```

EDASS ist jetzt aktiviert. Mit BASIC kann weiterhin wie gewohnt gearbeitet werden. EDASS und BASIC teilen sich den verfügbaren Speicher. Die Funktionen von EDASS werden über eine Befehlserweiterung aufgerufen.

Auf der Diskette wird noch eine zweite Version des Programms mitgeliefert, die mit RUN "EDASS 128.2" gestartet wird. Die Version ist in der Bedienung identisch und weist nur Unterschiede in der Einbindung in den C128 auf. Genaueres erfahren Sie in Kapitel 20 im Abschnitt über technische Informationen.

Um EDASS wieder zu verlassen, gibt es wieder mehrere Wege. Der spezielle EDASS-Befehl !COLD deaktiviert den Assembler und gibt den gesamten Speicher für BASIC frei.

Ein BASIC-Programm wird dabei nicht gelöscht, die Variableninhalte gehen jedoch verloren. Der zweite Weg wird durch Drücken der <RESET>-Taste

erreicht. Da die EDASS-Diskette bootfähig ist, muß sie natürlich zuvor aus dem Laufwerk genommen werden, sonst startet EDASS wieder neu. Vor dem Ausschalten des Computers entfernen Sie natürlich EDASS ebenfalls.

Übrigens sollten Sie sich von der Originaldiskette eine Kopie machen , um sie nicht dem Risiko einer unabsichtlichen Zerstörung auszusetzen.

Kapitel 14

Der Editor

Die grundlegende Arbeit mit EDASS

Damit Sie alle Erklärungen der folgenden Kapitel nachvollziehen können, finden Sie aus dem Assemblerkurs in Kapitel 2 ein ausführliches Beispiel, das die Arbeit mit EDASS demonstriert. Hier nochmals in Stichpunkten der Ablauf zur Erstellung eines Programms:

1. Ein neues Programm mit

   `!BEGIN "PROGRAMMNAME"` beim Editor anmelden.

2. Den Editor mit

   `!EDIT "PROGRAMMNAME"` aufrufen.

3. Das Programm zeilenweise eingeben und eventuell Korrekturen vornehmen.

4. Den Editor mit einem Kleinerzeichen ("<") oder einem Pfeil nach links (←) wieder verlassen.

5. Den Assembler mit dem Befehl

   `!ASSEMBLER "PROGRAMMNAME"` starten.

6. Das erzeugte Maschinenprogramm mit

   `SYS (Startadresse)` oder `!GO (Startadresse)` ablaufen lassen.

Die Eingabemaske

Nach Aufruf des Editors erscheint auf dem Bildschirm die Eingabemaske. Der C128 verfügt über zwei Bildschirmdarstellungen, mit 40- oder 80-Zeichen pro Zeile. Damit Benutzer, die nur den 40-Zeichen-Modus nutzen können, in

den Genuß breiterer Zeilen kommen, wurde EDASS mit einem Pseudo-80-Zeichen-Editor ausgerüstet. Erreicht der Cursor den rechten Rand des Bildschirms, wandert der sichtbare Programmtext am linken Rand hinaus, und von rechts wird neuer Text nachgeschoben. Entsprechendes geschieht bei Erreichen des rechten Randes. Auf dem Bildschirm ist also immer ein 40 Zeichen breites Fenster der 80-Zeichen-Zeile zu sehen.

Die Zeilenlänge des Pseudo-Editors kann mit dem MOD-Befehl variiert werden. Aus dem Pseudo-80-Zeichen-Editor wird beispielsweise ein Pseudo-60-Zeichen-Editor. Bei 40 Zeichen Breite bleibt der Bildschirminhalt konstant und wird nicht verschoben.

Im eingebauten 80-Zeichen-Modus des C128 ist natürlich immer die ganze Zeile zu sehen, und es bewegt sich kein Text beim Erreichen der Bildschirmgrenzen. Beide Bildschirmdarstellungen unterscheiden sich nur in den Informationen der Statuszeile. In Abb. 14.1 sehen Sie ein Bild der Eingabemaske.

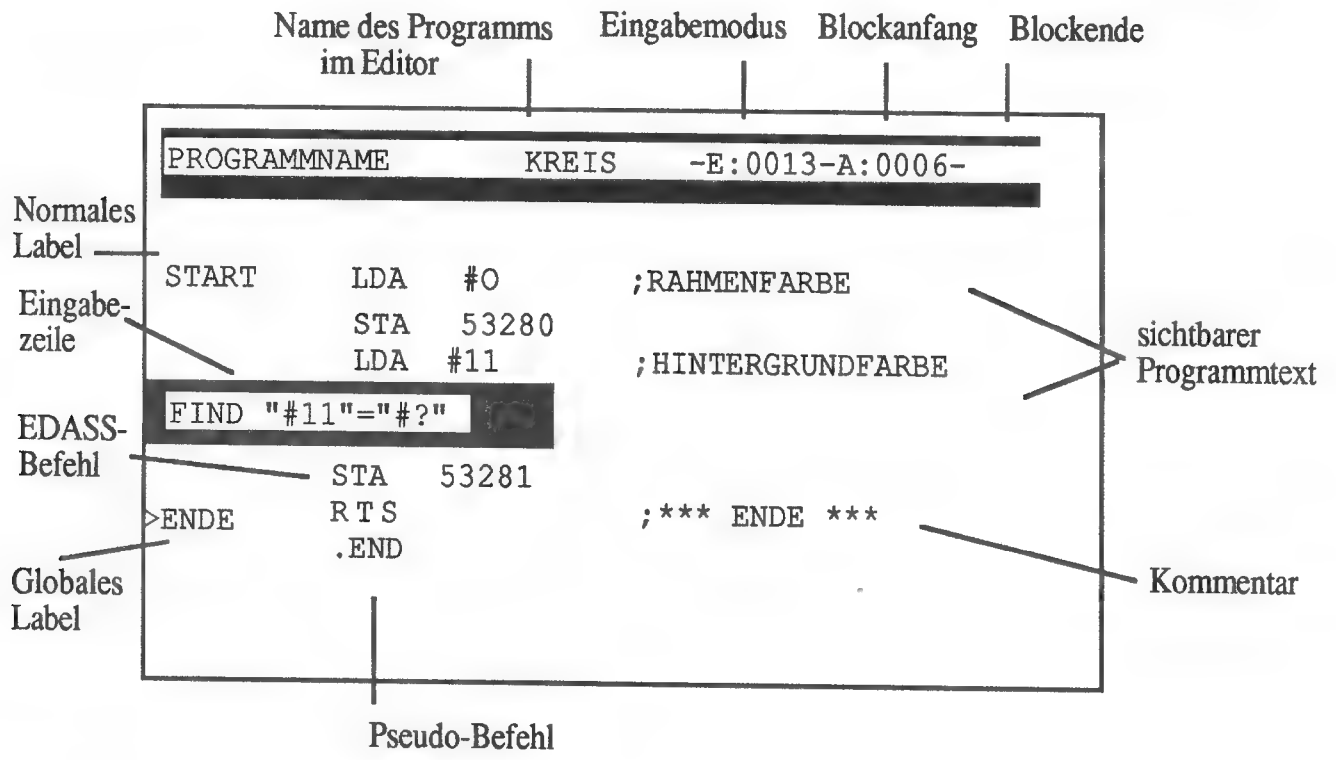


Abb. 14.1: Eingabemaske

Zum Schluß noch ein Tip: Schalten Sie im 80-Zeichen-Modus Ihren C128 auf den FAST-Modus. Die langsame Ansteuerung des 80-Zeichen-Videochips wird damit wieder ausgemerzt.

Die Eingabezeile

In der Mitte der Eingabemaske sehen Sie eine durch Negativdarstellung und eine besondere Farbe hervorgehobene Zeile. Diese Zeile dient als Eingabezei-

le. Der Cursor wartet auch dort auf Ihre Eingaben. In der Zeile darf der Cursor frei bewegt werden, kann jedoch diese nicht verlassen. Der sichtbare Programmtext wird ober- und unterhalb der Zeile dargestellt. Die Eingabezeile dient in erster Linie zur Eingabe von Programmzeilen. Aber auch die Befehle von EDASS können hier eingegeben werden. Mit Hilfe des MOD-Befehls kann die Markierung der Eingabezeile frei gewählt werden, Positiv- und Unterstreichdarstellung sowie andere Farben sind erreichbar.

Die Statuszeile

In dieser Zeile, die sich am oberen Bildschirmrand befindet, werden wichtige Informationen über den Zustand des Editors angezeigt. Durch die breiteren Zeilen im 80-Zeichen-Modus enthält die Statuszeile hier auch mehr Informationen.

Zunächst zum 40-Zeichen-Modus: Ganz links erscheint der Name des Programms, das Sie gerade bearbeiten; als nächstes wird der Eingabemodus angezeigt, ein "E" für Einfügen, ein "A" für Ändern und ein "K" für Kommentareingabe. Direkt daneben sehen Sie die Zeilennummer der augenblicklichen Eingabeposition, im Änderungsmodus die Zeile, die gerade bearbeitet wird, im Einfügemodus die Zeile unter der Eingabezeile. Noch weiter rechts steht die Zeilennummer, die als Blockanfang bzw. als Blockende festgesetzt wurde. Ist keine Zeile definiert, erscheint statt dessen ein leerer Bereich.

Jetzt die Unterschiede bei der 80-Zeichen-Darstellung: Der Eingabemodus wird voll ausgeschrieben, "EINFUEGEN" statt "E", "AENDERN" statt "A" und "KOMMENTAR" statt "K". Außer der Eingabezeile erscheint, getrennt durch einen Punkt (.), die Spaltenposition des Cursors. Die Statuszeile wird auch zur Ausgabe einzeiliger Meldungen verwendet, wie z.B. für Fehlermeldungen oder das Endergebnis einer mathematischen Berechnung.

Die Tastenbelegung

Im Editor haben viele Tasten und Tastenkombinationen eine neue Funktion erhalten:

<RETURN>

Wie beim normalen Bildschirmeditor wird mit dieser Taste eine neue Programmzeile übernommen. Bis dahin ist ein eingegebener Zeileninhalt nicht im Programm gespeichert!

<HOME>

Setzt den Cursor an den linken Rand des Bildschirms. Im 40-Zeichen-Modus ist dies der Anfang einer Pseudo-80-Zeichenzeile.

<CLR>

Löscht die Eingabezeile und setzt den Cursor wie nach <HOME >an den linken Rand des Bildschirms.

<CURSOR UP>

Ermöglicht das Aufwärtsfahren im Programmtext. Der Text bewegt sich dabei von oben nach unten und verschwindet am unteren Bildschirmrand. Im Änderungsmodus werden immer neue Zeilen in der Eingabezeile ausgegeben. Im Einfügemodus bleibt der Inhalt der Eingabezeile unverändert. Sie können also eine Zeile eintippen, mit den Cursortasten die gewünschte Programmposition anfahren und dann zum eigentlichen Einfügen die <RETURN>-Taste drücken.

<CURSOR DOWN> und <LINE FEED>

Dient dem Abwärtsfahren im Programmtext. Der Text bewegt sich dabei von unten nach oben.

<<> und <←>

Das Kleinerzeichen (<) und der Pfeil nach links (←) dienen als Abkürzung für den EXIT- und JUMP-Befehl (siehe Kapitel 15). Diese beiden Zeichen wurden gewählt, da sie bei ASCII- und DIN-Tastatur auf der gleichen Taste liegen. Ein Druck auf die Taste im ASCII-Modus ergibt einen Pfeil nach links und bei DIN-Tastatur das Kleinerzeichen.

<SHIFT> und <RETURN>

Dient dem Umschalten zwischen den zwei Eingabemodi "Einfügen" und "Ändern". Der Quelltext wird nach dem Tastendruck sofort auf die entsprechende Form gerückt. Beim Umschalten auf den Änderungsmodus bedeutet dies, daß die Zeile unter der Eingabezeile in die Eingabezeile geschoben wird. Beim Umschalten auf den Einfügmodus erfolgt genau das Umgekehrte. Die Einga-

bezeile wird gelöscht, und die Programmzeile, die geändert werden sollte, erscheint wieder im alten Zustand unter der Eingabezeile.

<TAB> und <CTRL> und <I>

Ermöglicht das schnelle Nach-rechts-Bewegen in der Eingabezeile. Der Cursor springt dabei von Formatposition zu Formatposition. Innerhalb des Kommentars bewegt er sich in 8er-Schritten nach rechts. Die letzte damit erreichbare Position ist der rechte Rand der Eingabezeile. Steht der Cursor z.B. am linken Rand und Sie drücken dreimal auf die TAB-Taste, befindet er sich an der Formatposition des Adreßteils.

<SHIFT> sowie <CTRL> und <X>

Hat dieselbe Bedeutung wie TAB, jedoch bewegt sich der Cursor von rechts nach links.

<CTRL> und <A> (Anfang setzen)

Dient zum Festsetzen des Blockanfangs. Es wird dabei die Zeilennummer der Eingabezeile, also die Nummer, die in der Statuszeile angezeigt wird, als Blockanfang übernommen. Nach dem Tastendruck verschwindet sofort der Freiraum in der Statuszeile, und es erscheint statt dessen die Zeilennummer. Um einen anderen Blockanfang zu setzen, drücken Sie einfach ein zweites Mal auf die Tasten <CTRL> und <A>. Die Blockmarken können beim INSERT-, ERASE- und FIND-Befehl zum Begrenzen eines Programmbereichs verwendet werden.

<CTRL> und <B> (Block löschen)

Löscht beide Blockmarken. Es erscheinen in der Statuszeile wieder zwei Freiräume.

<CTRL> und <C> sowie <RUN/STOP> (Copy)

Ermöglicht im Änderungsmodus, die Zeile, die geändert wird, noch einmal in der gespeicherten Form auszugeben. Bei Tippfehlern kann damit der alte Inhalt der Zeile zurückgerufen werden. Bei Verwendung im Einfügemodus gibt diese Taste die Zeile unter der Eingabezeile in der Eingabezeile ein zwei-

tes Mal aus. Es ist dann leicht möglich, diese Zeile mittels der Cursortasten an einen anderen Ort zu transportieren und dort in das Programm einzufügen. Auf diese Weise kann eine einzelne Zeile leicht kopiert werden.

<CTRL> und <D> (Down)

Ermöglicht das "Abwärtsblättern" im Programmtext. Der Quelltext wandert so weit nach oben, daß die letzte sichtbare Zeile am unteren Rand jetzt am oberen Rand zu sehen ist.

<CTRL> und <E> (Ende setzen)

Dient dem Festsetzen des Blockendes. Es gilt dabei das gleiche wie bei der Definition des Blockanfangs.

<CTRL> und <K> (Kommentar löschen)

Löscht den durch einen Kommentar belegten Bereich und setzt den Cursor an den Anfang des Kommentarfeldes. Der Strichpunkt zur Kommentarmarkierung wird auch gelöscht. Für einen neuen Kommentar muß dieser deshalb neu eingegeben werden.

<CTRL> und <L> (Label löschen)

Löscht den durch ein Label belegten Formatbereich in der Eingabezeile und setzt den Cursor an die Eingabeposition eines Labels.

<CTRL> und <O> (Operand löschen)

Löscht den durch einen Operanden belegten Bereich in der Eingabezeile und setzt den Cursor an den Anfang des Operandenfeldes.

<CTRL> und <U> (Up)

Ermöglicht das "Aufwärtsblättern" im Programmtext. Der Programmtext wandert dabei von oben nach unten.

<CTRL> und <W> (Wechseln des Eingabemodus)

Diese Tastenkombination schaltet in einen dritten verfügbaren Eingabemodus. In der Eingabezeile wird der Cursor auf die Formatposition des Kommentars gesetzt. Dort wird das Kommentarzeichen (";") ausgegeben. Der Kommentartext darf jetzt direkt folgend eingegeben werden. Mit diesem Modus kann beispielsweise ein Programm ohne Kommentar eingegeben und getestet werden und anschließend schnell nachkommentiert werden. Durch Löschen des vorgegebenen Kommentarzeichens kann natürlich der Zeileninhalt auch ohne Kommentareingabe geändert werden. Auch das Kommentarzeichen und der Kommentarinhalt werden erst mit der <RETURN>-Taste übernommen. Vorher kann mit den Cursortasten eine neue Zeile ausgewählt werden.

Außer diesen neuen Tastenfunktionen stehen die vom normalen Editor bekannten ESC-Sequenzen zur Verfügung. Diese beziehen sich jetzt natürlich nur auf die Eingabezeile und nicht auf den vollen Bildschirm. Hier eine Übersicht der Kombinationen:

<ESC> und <A>

Der automatische Einfügemodus des Editors wird eingeschaltet. Es werden so lange Zeichen eingefügt, bis die Tasten <ESC> und <C> gedrückt werden.

<ESC> und <C>

Schaltet den automatischen Einfügemodus wieder aus.

<ESC> und <D> oder <I> oder <V> oder <W>

All diese ESC-Sequenzen löschen die ganze Eingabezeile.

<ESC> und <E>

Schaltet den Cursor von Blinken auf Konstantanzeige.

<ESC> und <F>

Schaltet den Cursor zurück auf blinkende Anzeige.

<ESC> und <G>

Erlaubt das akustische Signal <CTRL> und <G>.

<ESC und <H>

Unterbindet das akustische Signal <CTRL> und <H>.

<ESC> und <J>

Setzt den Cursor an den Anfang der aktuellen Zeile. Diese Tastenkombination entspricht in der Wirkung der HOME-Taste.

<ESC> und <K>

Setzt den Cursor auf die Position nach dem letzten Zeichen in der Eingabezeile.

<ESC> und <N>

Schaltet den Bildschirm auf positive Darstellung (Gegenstück zu <ESC> und <R>).

<ESC> und <O>

Schaltet alle Cursormodi wie Blinken, Unterstreichen usw. ab.

<ESC> und <P>

Löscht die Eingabezeile von Anfang bis zur Cursor-Position.

<ESC> und <Q>

Löscht die Eingabezeile ab der Cursor-Position.

<ESC> und <R>

Schaltet den Bildschirm auf Invers-Darstellung (Gegenstück zu <ESC> und <N>).

<ESC> und <S>

Schaltet Blockdarstellung für den Cursor ein (Gegenstück zu <ESC> und <U>).

<ESC> und <U>

Schaltet Strichdarstellung für den Cursor ein (Gegenstück zu <ESC> und <S>).

<ESC> und <X>

Dient zum Umschalten zwischen den beiden Bildschirmmodi des C128. Die Umschaltung kann im Editor erfolgen, jedoch geht dabei der Inhalt der Eingabezeile verloren.

Folgende Sequenzen wurden außer Kraft gesetzt:

<ESC> und <L> oder <M> oder <T> oder <Y> oder <Z>

Eingabeformat einer Zeile

Bei der Eingabe einer Zeile muß darauf geachtet werden, daß der Assemblerbefehl von einem vorangehenden Label bzw. einer nachfolgenden Adresse durch eine Leerstelle abgegrenzt ist. Dies ist z.B. notwendig, damit der Editor nicht aus dem Label "START" den Befehl "STA" herausliest. Wird als Adressierungsart der Akkumulator verwendet (z.B. LSR A), darf das "A" zur Kennzeichnung entfallen. Ein Kommentar wird durch einen Strichpunkt markiert. Es sind auch reine Kommentarzeilen erlaubt. Sonst kann für die Eingabe ein beliebiges Format gewählt werden.

Hat der Editor keinen Befehl in der Zeile gefunden, wird die Fehlermeldung

```
?MNEMONIK NICHT VORHANDEN
```
ausgegeben.

Einfügen einer Zeile

Nach Aufruf des Editors ist bereits der Einfügemodus eingestellt. Während des Arbeitens können Sie den Modus auch mit den Tasten <SHIFT> und <RETURN> einschalten. Die Eingabezeile symbolisiert jetzt die Lücke im Quelltext, in der eine neue Zeile eingefügt werden kann. Nachdem Sie die Zei-

le eingegeben und <RETURN> gedrückt haben, wird die Zeile verarbeitet und formatiert über der Eingabezeile wieder ausgegeben. Die neue Einfügeposition befindet sich direkt unter der zuletzt eingegebenen Zeile. Es ist somit möglich, durch fortlaufende Eingabe einzelner Zeilen einen fortlaufenden Programmtext einzugeben.

Ändern einer Zeile

Der Änderungsmodus wird durch die Tasten <SHIFT> und <RETURN> aufgerufen. Die Zeile, die geändert werden soll, wird jetzt in der Eingabezeile angezeigt und kann wie eine neu eingetippte Zeile editiert werden. Nach Eingabe von <RETURN> wird die Zeile über der Eingabezeile ausgegeben. Die nächste Zeile, die geändert werden kann, wird automatisch von unten in die Eingabezeile geschoben.

Am Programmende kann der Änderungsmodus auch zur Eingabe neuer Zeilen verwendet werden. Es wird sozusagen die Zeile nach der letzten Zeile geändert.

Löschen einer Zeile

Zum Löschen einer Zeile wählen Sie mit den Tasten <SHIFT> und <RETURN> den Änderungsmodus.

Bewegen Sie jetzt die Eingabezeile mit den Cursortasten so lange, bis die Programmzeile, die gelöscht werden soll, in der Eingabezeile steht. Löschen Sie die Eingabezeile mit der <CLR>-Taste, und drücken Sie dann die <RETURN>-Taste. Die Zeile ist nun im Speicher gelöscht. Die nachfolgende Programmzeile steht in der Eingabezeile, um geändert oder auch wieder gelöscht zu werden.

Kapitel 15

Neue Befehle

Allgemeine Hinweise

Hier werden ein paar allgemeine Erklärungen gegeben, die für alle Befehle gelten. Ein EDASS-Befehl wird zur Kennzeichnung durch ein Ausrufezeichen (!) eingeleitet. Dieses Zeichen kann aber mit dem MOD-Befehl frei gewählt werden. Ein Befehl muß allein auf einer Bildschirmzeile stehen, darf also nicht mit BASIC-Befehlen gemischt werden. Die Befehle dürfen auch nicht in einem BASIC-Programm verwendet werden.

Jeder Befehl kann durch SHIFTen eines Zeichens abgekürzt werden (siehe Anhang E). Weiter haben Sie die Möglichkeit, alle neuen Befehle vom Editor aus aufzurufen. Der Befehl samt Ausrufezeichen muß dazu einfach auf die Eingabezeile geschrieben werden. Die Befehle DISPLAY, GO, LBL, LBL$, LIST und TYPE bilden hier die einzigen Ausnahmen. Sie werden im Editor nicht ausgeführt.

Wird im Befehl eine Zahleneingabe verlangt, kann an dieser Stelle ein beliebiger mathematischer Ausdruck stehen. Alle Rechenoperationen des Assemblers stehen dabei zur Verfügung (siehe Kapitel 17).

Damit Sie nicht immer den vollständigen Namen eines Programms eintippen müssen, haben Sie Jokerzeichen zur Verfügung. Diese entsprechen denen der Diskettenstation, das Fragezeichen (?) für ein unbekanntes Zeichen und das Sternchen (*) für das Ende einer Zeichenfolge.

```
!EDIT "TEST-TEXT" oder kurz !EDIT "TEST*"
```

Um ein Programm auf der Diskette zu überschreiben, muß ein Klammeraffe (@) bzw. bei deutschem Zeichensatz ein Paragraphenzeichen (§) dem Programmnamen vorangestellt werden. Dies gilt für alle Befehle, die auf die Diskette zugreifen.

```
!SAVE   "TEST"  Speichern ohne Überschreiben
!SAVE @ "TEST"  Speichern mit Überschreiben
!PULL $ "TEST"  Speichern mit Überschreiben
```

Einteilung der Befehle

Alle Befehle können in fünf Gruppen zusammengefaßt werden. Jede Gruppe hat einen speziellen Wirkungsbereich. Diese Zusammenstellung hilft Ihnen sicher, für bestimmte Aufgaben den richtigen Befehl zu finden.

1. Zu den Befehlen, die sich auf ein ganzes Programm beziehen, gehören BEGIN, EDIT, RENAME, LOAD, SAVE, IMPORT und LIST. Außerdem ERASE und INSERT, die ein ganzes Programm löschen bzw. einfügen können. Auch der DISPLAY-Befehl, der eine Liste aller im Speicher befindlichen Programme ausgibt, gehört in diese Gruppe.

2. Das Programm wird mit den Befehlen EDIT, ERASE, INSERT, FIND, JUMP und EXIT editiert.

3. Die Verarbeitung des Objektcodes übernehmen die Befehle ASSEMBLER, REASS, BYTE und GO. Wie Sie sehen, fällt in diese Gruppe der Assembler und Reassembler.

4. Für die Nachbearbeitung oder den Aufbau einer Labelliste sind die Befehle LET, LBL, LBL$, PULL, PUSH, CLEAR und KILL verantwortlich. Es können mit diesen Befehlen z.B. Labellisten, die der Assembler produziert hat, für den Reassembler aufbereitet werden.

5. Die noch verbleibenden Befehle =, $=, =, %=, MOD, NEW, STORE, TYPE und COLD übernehmen spezielle Aufgaben.

Beschreibung der EDASS-Befehle

Es folgt nun eine Erklärung der Befehle im einzelnen. Ausdrücke in kursiver Schrift müssen bei der Befehlsverwendung nicht unbedingt mit eingegeben werden. Für Geräteadressen wird z.B. automatisch die Adresse 8 angenommen.

Wird in der Beschreibung als Parameter "Zeilenbereich" verlangt, muß ein Programmbereich ausgewählt werden. Der erste Weg erfolgt über die direkte Angabe der Zeilennummer von Anfangs- und Endzeile. Beide Werte müssen durch ein Komma getrennt werden.

Entfällt die Anfangszeile, wird ersatzweise die Zeile des Programmanfangs eingesetzt, für die Endzeile entsprechend die Zeile des Programmendes. Bei Angabe einer einzelnen Zeilennummer wird nur diese Zeile bearbeitet. Hier einige Anwendungsbeispiele:

`!FIND "TEST",10,20`	von Zeile 10 bis Zeile 20
`!INSERT ,100`	von Anfang bis Zeile 100
`!ERASE 200,`	von Zeile 200 bis Ende
`!INSERT 45`	nur Zeile 45

Einige spezielle Bereiche werden durch Buchstaben gewählt. Ein "B" wählt den durch Blockanfang und Blockende gekennzeichneten Bereich. Fehlt eine Blockmarke oder ist die Endmarke kleiner als die Anfangsmarke, erscheint die Fehlermeldung

```
?FEHLERHAFTE BLOCKMARKEN
```

Durch das Zeichen "A" wird der Bereich vom Programmanfang bis zur Eingabezeile gewählt und mit "E" von der Eingabezeile bis zum Programmende. Zu beachten ist, daß im Einfügemodus die Zeilennummer der Eingabezeile der Zeile unter der Eingabezeile entspricht. Wieder einige Befehlsbeispiele:

`!FIND "TEST",B`	Blockbereich
`!INSERT E`	Eingabezeile bis Programmende
`!ERASE A`	Anfang bis Eingabezeile

Beim FIND-Befehl darf die Zeilenangabe auch ganz entfallen. Es wird dann das gesamte Programm durchsucht.

! = (math. Ausdruck)

Dieser Befehl wertet einen beliebigen mathematischen Ausdruck aus und gibt das Ergebnis in dezimaler Form auf dem Bildschirm aus. Im Editor wird das Ergebnis in der Statuszeile dargestellt.

Im Ausdruck selbst können alle Operationen verwendet werden, die EDASS anbietet (siehe Kapitel 17). Zahlensysteme können beliebig gemischt werden. Labels dürfen ebenfalls im Ausdruck enthalten sein:

```
!=(800+$ab-%101)*"a"
!=2*test+anfang-100
```

!$ = (math. Ausdruck)

Dieser Befehl hat dieselbe Funktion wie der =-Befehl, gibt jedoch das Ergebnis nicht dezimal, sondern im Hexadezimalsystem aus. Zur Kennzeichnung steht vor dem Zahlenwert ein Dollarzeichen ($).

!@ = (math. Ausdruck) oder !§ = (math. Ausdruck)

Dieser Befehl wertet auch wieder einen mathematischen Ausdruck aus. Das Ergebnis wird im Oktalsystem (System zur Basis 8) ausgegeben. Bei ASCII-Tastatur ist das Befehlswort ein Klammeraffe und bei DIN-Tastatur ein Paragraphenzeichen.

!% = (math. Ausdruck)

Dieser Befehl hat dieselbe Funktion wie die vorangegangenen Befehle, jedoch wird das Ergebnis binär ausgegeben. Die Zahl umfaßt grundsätzlich 16 Stellen.

!ASSEMBLER "(Programmname)", (Geräteadresse)

Durch diesen Befehl starten Sie den Assembliervorgang eines Programms. Der Assembler schaut im Speicher nach, ob sich das Programm dort befindet. Findet er den Namen nicht, nimmt er an, daß das Programm auf Diskette gespeichert ist und liest den Quelltext direkt von Diskette. Das Programm wird dabei nicht in den Computer geladen. Wird der Befehl vom Editor aus aufgerufen, darf der Programmname entfallen. Es wird dann das augenblicklich bearbeitete Programm assembliert. Beim Aufruf des Befehls aus dem Editor steht am Ende des Assemblierens die Eingabezeile wieder bei der Zeile Null, also am Anfang des Programms.

Stellt der Assembler im Programm einen Fehler fest, gibt er die Fehlermeldung, den Programmnamen und die Zeilennummer, in der der Fehler aufgetreten ist, auf dem Bildschirm aus und bricht die Assemblierung ab. Als Schlußmeldung erscheint die Endadresse und der Text "ENDE DER ASSEMBLIERUNG!".

Die Meldung "PASS-DIFFERENZ!" weist darauf hin, daß der Assembler im ersten und zweiten Paß einen unterschiedlich langen Objektcode erzeugt hat. Fehlerhafte Zeilen oder Befehle mit Zero-Page-Adressierung können Ursache hierfür sein. Genaues finden Sie in Kapitel 18, wo die Fehlermeldungen behandelt werden. Der Assembliervorgang kann jederzeit durch die <RUN/-STOP>-Taste abgebrochen werden.

!BEGIN "(Programmname)"

Mit BEGIN fangen Sie ein neues Programm im Speicher an. Dies bedeutet nichts anderes, als daß der Name in die Programmliste des Editors eingetragen

wird und damit ein neues Programm zum Edieren bereitsteht. Der BEGIN-Befehl ist grundsätzlich der erste Schritt, um ein neues Programmm einzugeben.

Befindet sich bereits ein Programm mit demselben Namen im Speicher, wird die Meldung

```
?PROGRAMM BEREITS IM SPEICHER
```

ausgegeben. Der Speicher kann zehn Programme aufnehmen. Versucht man, ein elftes zu beginnen, erscheint die Fehlermeldung:

```
?ZU VIELE PROGRAMME
```

!BYTE "(Programmname)",(Anfangsadr.),(Endadr.)

Dieser Befehl wandelt einen Speicherbereich des Computers in eine editierfähige Tabelle um. Zu diesem Zweck wird ein neues Programm mit dem im Befehl angegebenen Namen begonnen. In dieses Programm werden die Inhalte der Speicherzellen mit Hilfe des Pseudobefehls .BYTE in der Form einer Tabelle eingetragen.

Eine Zeile besteht dann aus dem .BYTE-Befehl und jeweils acht hexadezimalen Werten. Die Auswahl der Speicherbank muß zuvor mit dem BASIC-Befehl BANK geschehen. Ersatzweise kann auch die letzte Bankwahl des Assemblers verwendet werden. In Kombination mit dem REASS-Befehl kann der BYTE-Befehl benutzt werden, große Tabellen ebenfalls in ein editierfähiges Programm umzuwandeln:

```
!BYTE "TEST",$1000,$10FF
```

!BYTE "(Programmname)",D*(Geräteadr.)*,(Anfangsadr.),(Endadr.)

Gegenüber dem zuvor beschriebenen BYTE-Befehl wird bei dieser Befehlsform der Speicherinhalt des Diskettenlaufwerks ausgelesen und in eine editierfähige Tabelle verwandelt:

```
!BYTE "P.TAB",D,$C000,$C100
```

!BYTE "(Programmname)","(Dateiname)",*(Geräteadr.)*

Gegenüber den zwei zuvor beschriebenen Formen des Befehls gewinnt dieser Befehl die Tabellenwerte aus einer Diskettendatei. Die Startadresse des Pro-

gramms muß, wie es bei Programmdateien üblich ist, am Anfang der Datei eingetragen sein:

```
!BYTE "P.TAB","LOADER",9
```

!CLEAR

Der CLEAR-Befehl löscht alle im Speicher befindlichen Labels. Als einziger Befehl löscht !ASSEMBLER beim Aufruf selbständig die Labels.

!COLD

Dieser Befehl dient zum Verlassen von EDASS. Der Assembler wird desaktiviert, der gesamte Speicher wieder BASIC zur Verfügung gestellt. Ein eventuell vorhandenes BASIC-Programm bleibt erhalten, die Variableninhalte gehen jedoch verloren. Dieser Befehl kann später nicht mehr rückgängig gemacht werden! EDASS muß wieder neu geladen werden.

!DISPLAY

Mit diesem Befehl werden die Namen aller im Speicher befindlichen Programme ausgegeben. Es erscheinen die Namen in Anführungszeichen und zusätzlich die Zahl der Zeilen, die das Programm bereits enthält.

!EDIT "(Programmname)"

Mit dem EDIT-Befehl rufen Sie den Editor auf. Das angegebene Programm muß zuvor z.B. mit dem BEGIN-Befehl angemeldet werden. Der Befehl muß immer durchgeführt werden, um ein Programm zu erweitern oder zu ändern. Ist der Programmname im Speicher nicht vorhanden, wird die Fehlermeldung

```
?PROGRAMM NICHT IM SPEICHER
```

ausgegeben. Beim Befehlsaufruf im Editor wird das neue Programm bereitgestellt, auf den Einfügemodus geschaltet, und die Blockmarken werden gelöscht.

!ERASE (Zeilenbereich)

Diese Befehlsform kann nur im Editor ausgeführt werden. Sie ermöglicht es, einen Teil des augenblicklich bearbeiteten Programms zu löschen. Der Bereich wird durch die Zeilenangabe begrenzt.

`!ERASE 50,78`	Zeile 50 bis 78
`!ERASE E`	Eingabezeile bis Programmende
`!ERASE B`	Blockbereich
`!ERASE 50,`	Zeile 50 bis Programmende

!ERASE "(Programmname)"

Diese Befehlsform löscht ein komplettes Programm. Einer der 10 Programmplätze wird dabei wieder frei. Ist der angegebene Name nicht im Speicher zu finden, wird der Fehler

```
?PROGRAMM NICHT IM SPEICHER
```

ausgegeben.

Wird der Befehl im Editor ausgeführt und entspricht der Programmname dem des augenblicklich bearbeiteten Programms, wird das Programm gelöscht und der Editor wie durch einen EXIT-Befehl verlassen.

!EXIT

Mit dem EXIT-Befehl verlassen Sie den Editor. Der Bildschirm wird gelöscht, und der Computer arbeitet wieder mit seinen normalen Eingaberoutinen. Mit dem EDIT-Befehl kann danach wieder der Editor erreicht werden. Da dieser Befehl sehr häufig benötigt wird, kann er durch einen Pfeil nach links (←) oder Kleinerzeichen (<) abgekürzt werden. Diese beiden Zeichen wurden gewählt, da sie auf der ASCII- und DIN-Tastatur auf derselben Taste liegen.

!FIND"(Suchstring)",*(Zeilenbereich)*

Dieser Befehl durchsucht das Programm, das sich gerade im Editor befindet, nach einer Zeichenfolge. Dies können Labelnamen, Befehle, Adressen oder Teile des Kommentars sein. Wurde eine Zeichenfolge gefunden, wird die betreffende Zeile in der Eingabezeile ausgegeben und der Cursor an den Anfang der gefundenen Zeichenfolge gesetzt. In der Statuszeile erscheint dann die entsprechende Zeilennummer und die Frage:

```
SUCHE FORTSETZEN?
```

Der Computer wartet jetzt, bis Sie die Taste "J" (ja) oder RETURN drücken, um die Suche fortzusetzen. Jede andere Taste beendet die Suche. In diesem Fall können Sie die soeben gefundene Programmstelle editieren.

War die Suche erfolglos, erscheint in der Statuszeile die Meldung

```
?NICHTS GEFUNDEN
```

Im Suchstring kommt dem Fragezeichen (?) eine besondere Aufgabe zu. An dieser Stelle darf im Text ein beliebiges Zeichen stehen. Bei "A?C" findet der Befehl z.B. "ABC", "AXC" oder ADC".

`!FIND "ERR"`	ganzes Programm
`!FIND "E?R",,58`	Programmanfang bis Zeile 58
`!FIND "ERR",B`	Blockbereich
`!FIND "ER?",A`	Programmanfang bis Eingabezeile

!FIND "(Suchstring)"="(Ersatzstring)",*(Zeilenbereich)*

Diese Befehlsform sucht eine Zeichenfolge und ersetzt diese dann auf Wunsch. Bevor die Ersatzzeichenfolge ins Programm eingebaut wird, erscheint die Frage:

```
ERSETZEN?
```

Zum Ersetzen muß die Taste "J" (ja) oder <RETURN> gedrückt werden. Jede andere Taste verneint die Frage. Im weiteren wird, wie oben beschrieben, gefragt, ob die Suche fortgesetzt werden soll usw.

```
!FIND "TEST"="TEXT",B
!FIND "$FFD2"="BSOUT"
```

!GO (Startadr.),*(ac-Inhalt)*,*(xr-Inhalt)*,*(yr-Inhalt)*

Dieser Befehl startet ein Maschinenprogramm im Computerspeicher. Gegenüber dem BASIC-Befehl SYS darf die Startadresse in verschiedenen Zahlenformaten vorliegen. Außerdem dürfen den Registern Startwerte übergeben werden.

Nach Abarbeitung des Maschinenprogramms gibt der Befehl die Inhalte des Akkumulators, X- und Y-Registers sowie des Statusregisters auf dem Bildschirm aus.

!GOD *(Geräteadr.)*,(Startadr.)

Mit diesem Befehl wird ein Maschinenprogramm im Speicher des Diskettenlaufwerks gestartet. Der Befehl ersetzt das Diskettenkommando "M-E".

!IMPORT "(Dateiname)",*(Geräteadr.)*

Dieser Befehl lädt ein Assemblerprogramm, das nicht im EDASS-Format abgespeichert wurde. Die Datei muß vom Typ "PRG" sein. Das ganze Programm selbst muß als ASCII-Text vorliegen. Die einzelnen Zeilen müssen durch einen Wagenrücklauf oder ein Byte mit dem Wert Null abgeschlossen werden. Mit dieser Vereinbarung können z.B. viele Programmtexte von einer Textverarbeitung übernommen werden. Sollte EDASS eine Zeile nicht verstehen – fehlende Mnemoniks oder unbekannte Pseudoanweisungen können die Ursache hierfür sein – wird der Text der Zeile als reine Kommentarzeile ins Programm eingebaut.

Der IMPORT-Befehl entfernt alle Ziffern am Zeilenanfang. Damit ist es auch möglich, Assemblerprogramme, die mit dem normalen BASIC-Editor eingegeben wurden und folglich auch Zeilennummern enthalten, in EDASS einzulesen.

Mit OPEN und CMD wird das Listing des Programms in eine Datei vom Typ "PRG" geleitet. Diese wird dann mit dem IMPORT-Befehl ausgelesen.Wie beim LOAD-Befehl erscheint auch hier eine Fehlermeldung, wenn kein ausreichender Programmspeicher vorhanden ist.

!INSERT (Zeilenbereich)

Dieser Befehl kann nur im Editor ausgeführt werden. Er ermöglicht es, einen Teil des augenblicklich bearbeiteten Programms an der Programmposition der Eingabezeile einzufügen.

Beim Befehlsaufruf darf sich die Eingabezeile, also die Position, an der eingefügt wird, nicht innerhalb des Bereichs befinden, der übertragen wird. Sollte dies doch der Fall sein, erscheint die Fehlermeldung:

```
?EINFUEGEN NICHT MOEGLICH
```

Nach dem Einfügen befindet sich die Eingabezeile noch an derselben Programmposition, also am Anfang des eingefügten Bereichs. Sollte der noch

zur Verfügung stehende Programmspeicher nicht mehr für den einzufügenden Text ausreichen, wird die Meldung ausgegeben:

```
?KEIN PROGRAMMSPEICHER MEHR
```

```
!INSERT 13,20        Zeile 13 bis 30 einfügen
!INSERT 17           Zeile 17 einfügen
!INSRET B            Blockbereich einfügen
```

!INSERT "(Programmname)",*(Geräteadr.)*

Diese Form des INSERT-Befehls kann nur im Editor ausgeführt werden. Er fügt jedoch ein komplettes Programm in das gerade bearbeitete ein. Auch hier führt ein nicht mehr ausreichender Programmspeicher zur Fehlermeldung:

```
?KEIN PROGRAMMSPEICHER MEHR
```

Sind der angegebene Name und der Name des augenblicklich bearbeiteten Programms identisch, erscheint die Fehlermeldung:

```
?EINFUEGEN NICHT MOEGLICH
```

Wurde der angegebene Name im Speicher nicht gefunden, wird das Programm nicht aus dem Speicher, sondern von der Diskette eingelesen und eingefügt.

Durch die RUN/STOP-Taste kann der Vorgang abgebrochen werden. Der bereits eingefügte Programmteil bleibt dabei erhalten. Er bleibt auch bestehen, wenn der Programmspeicher nicht mehr ausreicht und eine Fehlermeldung erschienen ist.

!JUMP (Zeilennr.)

Der JUMP-Befehl dient zum schnellen Erreichen einer bestimmten Programmposition, was vor allem bei längeren Programmen von großem Nutzen ist. Wird eine Zeilennummer angegeben, die größer ist als die Nummer der letzten Zeile, wird zum Ende des Programms gesprungen. Da der JUMP-Befehl häufig verwendet wird, kann er durch einen Pfeil nach links (←) oder Kleinerzeichen (<) abgekürzt werden.

Diese beiden Zeichen wurden gewählt, da sie bei ASCII- und DIN-Tastatur auf derselben Taste liegen. Zu bestimmten Programmpositionen, wie z.B. dem

Programmanfang, wird häufig gesprungen. Für diese Positionen können statt der Zeilennummer auch spezielle Buchstaben angegeben werden:

A	für den Programmanfang
E	für das Programmende
BA	für den Blockanfang
BE	für das Blockende

Wird versucht, zu einer nicht gesetzten Blockmarke zu springen, wird kein Sprung ausgeführt.

`!JUMP 89`	zur 89. Zeile springen
`<E`	zum Programmende springen
`<BA`	zum Blockanfang springen

!KILL (Labelname)

Dieser Befehl löscht ein einzelnes Label. Eine vom Assembler erzeugte Labelliste kann in Zusammenarbeit von LET- und KILL-Befehl aufbereitet werden. Zu beachten ist, daß der KILL-Befehl das Label nicht im Speicher löscht, sondern nur als gelöscht kennzeichnet. Speicherplatz kann also mit diesem Befehl nicht gewonnen werden.

!LBL

Dieser Befehl gibt alle im Speicher befindlichen Labels mit dezimaler Wertangabe aus. Es erscheinen alle globalen, im Hauptprogramm oder mit LET definierten Labels. Ein globales Label wird mit einem Größerzeichen (>) markiert. Die Ausgabe erfolgt auf den Bildschirm, kann aber selbstverständlich mit den Kommandos OPEN und CMD auf ein anderes Gerät umgeleitet werden. Der Befehl entspricht der Assembleranweisung .LBL.

!LBL$

Wie der zuvor beschriebene Befehl gibt auch der LBL$-Befehl die im Speicher befindlichen Labels aus, die Wertangaben erfolgen jedoch im Hexadezimalsystem.

!LET (Labelname)=(math. Ausdruck)

Mit LET können Sie Labels außerhalb des Assemblers definieren. Ist ein Label bereits vorhanden, kommt es zur Fehlermeldung

```
?LABEL BEREITS DEFINIERT
```

Der Befehl kann zum Aufbau einer Labelliste verwendet werden, die dann z.B. an den Reassembler übergeben wird.

```
!LET ABC=$45+3*TEST
!LET >DEF="X"
```

!LIST "(Programmname)",*(Anfangszeile),(Endzeile)*

Mit diesem Befehl geben Sie ein Programm, das sich im Speicher befindet, auf dem Bildschirm aus. Die Ausgabe kann mit OPEN und CMD auch auf ein anderes Gerät geleitet werden.

Wird weder eine Anfangs- noch eine Endzeile angegeben, wird das gesamte Programm ausgedruckt; werden Begrenzungen genannt, nur der damit gewählte Bereich. Fehlt die Endzeile, so erscheint der Bereich zwischen der angegebenen Anfangszeile und dem Programmende; bei fehlender Anfangszeile entsprechend der Bereich zwischen Programmanfang und Endzeile.

`!LIST "TEST"`	ganzes Programm
`!LIST "TEST",23,37`	Programmbereich von 23 bis 37
`!LIST "TEST",29,`	von Zeile 29 bis Programmende
`!LIST "TEST",,45`	Programmanfang bis Zeile 45

!LOAD "(Programmname)",*(Geräteadr.)*

Mit dem LOAD-Befehl können Sie durch SAVE zuvor abgespeicherte Programme wieder in den Computer einlesen. Ist ein Programm mit gleichem Namen bereits vorhanden, wird die Meldung ausgegeben:

```
?PROGRAMM BEREITS IM SPEICHER
```

Der LOAD-Befehl kann jederzeit durch die RUN/STOP-Taste abgebrochen werden. Das bis dahin eingeladene Programmstück bleibt erhalten und kann wie ein normales Programm editiert werden.

Da dies auch im Falle eines Diskettenfehlers gilt, kann so z.B. ein auf Diskette zerstörtes Programm wieder so weit wie möglich eingelesen werden. Ist kein freier Programmspeicher mehr vorhanden, bricht EDASS den Ladevorgang ab mit der Fehlermeldung:

```
?KEIN PROGRAMMSPEICHER MEHR
```

Auch in diesem Fall steht der bereits geladene Programmteil zur Verfügung.

!MOD(Editorbreite),
(40-Markierung),(80-Markierung),(Befehlszeichen)

Dieser Befehl verändert verschiedene Parameter von EDASS. Der erste Wert steuert die Zeilenbreite des Editors im 40-Zeichen-Modus. Diese kann zwischen 40 und 80 Zeichen variiert werden. Eine geringere Zeichenzahl hat den Nachteil, daß weniger Zeichen für jede Formatposition zur Verfügung stehen, jedoch wird der Zeileninhalt dadurch kompakter. Der zweite Parameter bestimmt das Aussehen der Eingabezeile im 40-Zeichen-Modus. Die unteren vier Bits (unteres Nibble) legen die Farbe der Eingabezeile fest, und mit dem höchstwertigen Bit (MSB) wird entschieden, ob die Zeile zusätzlich in Negativschrift erscheint.

Der dritte Parameter übernimmt die gleiche Funktion für den 80-Zeichen-Modus. Hier wird jedoch ein komplettes Attribut-Byte (siehe Anhang H) übergeben. Alle Darstellungsarten wie Farbe, Unterstreichen, Negativ und Blinken sind erreichbar. Bit 7, das zur Umschaltung zwischen den Zeichensätzen dient, bleibt jedoch unbeachtet.

Mit dem letzten Parameter verändern Sie das Einleitungszeichen der EDASS-Befehle. Dieses wird im ASCII-Code angegeben. Hier ist einige Vorsicht geboten. Im ASCII-Code gibt es neben den Steuercodes Zeichen, die nicht über die Tastatur eingegeben werden können. Alle Umlaute sind z.B. zweimal vertreten, aber nur an einer Stelle werden Sie durch die Tastatur angesprochen.

```
!MOD 60,7          60 Zeichen und gelbe Farbe
!MOD 60,7,34       ... und blaue Farbe/unterstrichen
!MOD 60,7,34,"#"   ...+Befehlszeichen "#"
```

!NEW (BASIC-Programm),(BASIC-Variablen)

Dieser Befehl teilt den Speicher des C128 neu zwischen BASIC und EDASS auf. Noch im Speicher befindliche BASIC- oder Assemblerprogramme werden gelöscht! Der erste Parameter bestimmt die Zahl der gewünschten BASIC-Programm-Bytes (min. 10 KByte, max. 54 KByte). Ein größerer Wert an dieser Stelle reduziert den Labelspeicher von EDASS. Mit dem zweiten Parameter wird Speicher für die BASIC-Variablen (mind. 3 KByte, max. 45 KByte) reserviert. Diese reduzieren den Programmspeicher von EDASS.

!PULL "(Dateiname)",*(Geräteadr.)*

Dieser Befehl liest eine mit dem EDASS-Befehl PUSH oder der Assembleranweisung .PUSH erzeugte Tabelle wieder von Diskette ein. Die Kennzeichnung ".P" wird automatisch an den Dateinamen angehängt.

!PUSH "(Dateiname)",*(Geräteadr.)*

Dieser Befehl speichert alle im Speicher befindlichen lesbaren Labels auf Diskette. Zu diesen Labels zählen alle globalen, im Hauptprogramm oder mit LET definierten Labels. Zur Unterscheidung wird dem Dateinamen am Ende ".P" angehängt.

!REASS "(Programmname)",(Anfangsadr.),(Endadr.)

Der REASS-Befehl ist das Gegenstück zum ASSEMBLER-Befehl. Er erzeugt aus einem Objektcode wieder ein editierfähiges Quellprogramm. Gegenüber einem Diassembler werden bei diesem Befehl jedoch alle absoluten und relativen Sprünge durch Labels ausgedrückt. In der Namensvergabe bezieht der Reassembler die im Speicher befindlichen Labels mit ein. Bei einem Operandenwert wird zuerst ein Label mit diesem Wert gesucht. Wird keines gefunden, wird bei programminternen absoluten und relativen Sprüngen ein Name aus der Hexadezimaldarstellung des Wertes und einem vorangestellten "L" erzeugt.

Bei Operanden, die auf Adressen außerhalb des Programms zeigen und bei denen kein entsprechendes Label vorhanden ist, wird der Zahlenwert direkt verwendet.

Wird beim Reassemblieren ein Byte gefunden, das nicht in einen Befehl des 6502- bzw. 8502-Prozessors zu übersetzen ist, wird der Wert mit Hilfe des .BYTE-Befehls als Minitabelle in das Quellprogramm eingebaut.

Darüber hinaus können auch bewußt bestimmte Bereiche des Programms in eine Tabelle gefaßt werden. Labels, die mit "BYTA" anfangen, bestimmen den Anfang der Tabelle und Labels mit "BYTE" das Ende. Die Beziehung zwischen beiden wird mit den nachfolgenden Zeichen hergestellt. Werden z.B. vor dem Reassemblieren die Definitionen

```
BYTATEST = $1300
BYTETEST = $1320
```

vorgenommen, wird der Programmbereich von $1300 bis $1320 in eine Tabelle aus .BYTE-Befehlen übernommen.

Eine weitere Tabelle könnte mit BYTAHILFE und BYTEHILFE definiert werden. Die externen Labels können von einem Aufruf des Assemblers stammen, mit dem EDASS-Befehl PULL eingelesen oder einzeln mit dem LET-Befehl definiert werden. Am Ende können selbstverständlich diese Labels zusammen mit den vom Reassembler erzeugten ausgegeben oder gespeichert

werden. Der REASS-Befehl beginnt selbständig ein neues Programm im Speicher unter dem im Befehl angegebenen Namen und schreibt das Quellprogramm dort hinein.

Der Objektcode, der reassembliert werden soll, steht im Computerspeicher und wird durch die Anfangs- und Endadresse begrenzt. Die Speicherbank muß zuvor mit dem BASIC-Befehl BANK eingestellt werden.

Wird der erzeugte Text für den Programmspeicher zu groß, erscheint die Fehlermeldung:

```
?KEIN PROGRAMMSPEICHER MEHR
```

Zum Schluß ein Beispiel, das alle Möglichkeiten des Befehls demonstriert. Reassembliert wird ein Stück aus dem Betriebssystem des C128.

Labeldefinitionen:

```
!LET CMPARE = $2BE          (Bank-Routine in Page 2)
!LET MMUCR = $FF00          (Speicherverwaltungsbaustein)
!LET BYTATAB = $F7F0        (Tabellenanfang)
!LET BYTETAB = $F7FF        (Tabellenende)
```

Die Befehle:

```
BANK 15                     (ROM-Bank wählen)
!REASS"ROM",$F7E3,$F80C     (Reassembler starten)
```

Das Endprodukt:

```
          .BANK   $0F
          *=      $F7E3
          PHA
          LDA     BYTATAB,X
          TAX
          PLA
          JMP     CMPARE
          LDA     BYTATAB,X
          RTS
BYTATAB   .BYTE   $3F,$7F,$BF,$FF,$16,$56,$96,$D6
          .BYTE   $2A,$6A,$AA,$EA,$06,$0A,$01,$00
          LDA     MMUCR
          STX     MMUCR
          TAX
          LDA     ($FF),Y
          STX     MMUCR
          RTS
          .END
```

!REASS "(Programmname)",D*(Geräteadr.)*,(Anfangsadr.),(Endadr.)

Wie der zuvor beschriebene REASS-Befehl reassembliert auch dieser Befehl einen Objektcode zu einem editierfähigen Quellprogramm. Bei dieser Befehlsform wird der Objektcode aus dem Speicher des Diskettenlaufwerks gelesen.

```
!REASS "P1",D,$C100,$C180
```

!REASS "(Programmname)","(Dateiname)",*(Geräteadr.)*

Wie die zwei zuvor beschriebenen REASS-Befehle erzeugt auch dieser Befehl aus einem Objektcode ein editierfähiges Quellprogramm. Bei diesem Befehl wird der Objektcode jedoch aus der angegebenen Programmdatei, die auf Diskette gespeichert ist, ausgelesen. Dabei ist vor allem zu beachten, daß das Programm nicht zu lang ist und dadurch die Kapazität des freien Speichers übersteigt. Die Startadresse, die bei der Programmdatei am Anfang stehen muß, wird auch als Startadresse für den Reassembliervorgang genommen.

```
!REASS "LOADER","PROG.LOAD"
```

!RENAME "(alter Name)"="(neuer Name")

Der Name eines im Speicher befindlichen Programms kann mit diesem Befehl geändert werden. Ist ein Programm mit dem alten Namen nicht im Speicher, wird die Fehlermeldung ausgegeben:

```
?PROGRAMM NICHT IM SPEICHER
```

Ist der neue Name bereits vorhanden, erscheint die Meldung:

```
?PROGRAMM BEREITS IM SPEICHER
```

!SAVE "(Programmname)",*(Geräteadr.)*

Dieser Befehl speichert ein Programm auf Diskette. Befinden Sie sich im Editor und möchten das Programm, das Sie gerade bearbeiten, abspeichern, kann der Name entfallen. Es wird dann das Programm im Editor gespeichert. Wird ein Name eingegeben, der nicht im Speicher zu finden ist , erscheint die Fehlermeldung:

```
?PROGRAMM NICHT IM SPEICHER
```

Der Speichervorgang kann jederzeit mit der RUN/STOP-Taste abgebrochen werden.

!STORE

Der STORE-Befehl sichert alle eingestellten Parameter von EDASS auf Diskette. Zu diesen zählen alle mit MOD veränderbaren Werte und die Speicheraufteilung, festgelegt mit NEW. Zusätzlich wird die augenblickliche Belegung der Funktionstasten gesichert. Alle Daten fließen in die Datei "PARAMETER".

Beim Start von EDASS wird diese wieder ausgelesen. Sie können so immer sofort mit Ihrer ganz individuellen Einstellung arbeiten.

!TYPE "(Dateiname)",*(Geräteadr.)*

Dieser Befehl gibt den ASCII-Text einer Datei vom Typ "PRG" aus. Wurden z.B. die Ausgaben des Assemblers in Dateien umgeleitet, kann dieser Befehl den Inhalt wieder darstellen. Natürlich kann auch hier die Ausgabe mit OPEN und CMD z.B. auf einen Drucker umgeleitet werden.

Kapitel 16

Der Assembler

Einige allgemeine Worte

Der Assemblerbestandteil von EDASS wird mit dem ASSEMBLER-Befehl gestartet. Der Assembler ist für die Sprache des 6502- bzw. 8502-Prozessors ausgelegt. Mnemoniks und Adreßformat sind vom MOS-Standard übernommen. Auch bei den Assembleranweisungen, den sogenannten Pseudo-Befehlen, wird weitgehend die übliche Syntax verwendet. Zur Adressierart sei noch gesagt, daß durch ein vorangestelltes Ausrufezeichen (!) bei der Adreßwahl, absolute Adressierung erzwungen werden kann: (LDA !123,X oder ADC !20). Diese Möglichkeit ist in einigen seltenen Fällen hilfreich.

Der Assembler arbeitet labelorientiert, d.h. Adressen, Werte und Programmpunkte müssen nicht mit ihren absoluten Werten angegeben werden, sondern können durch Labels vertreten werden.

Der Assembliervorgang läuft in zwei Durchgängen ab, dem ersten und zweiten Paß. Paß 1 dient zum Definieren der Labels, Paß 2 zum eigentlichen Errechnen des Maschinenprogramms. In diesem Schritt wird auch der Objektcode und das Assemblerlisting ausgegeben. Die Ausgabe kann auf unterschiedlichen Peripheriegeräten erfolgen, wie z.B. dem Drucker oder der Diskette.

Labels

Labels bzw. Marken, wie sie auch genannt werden, stehen stellvertretend für Adressen, Programmpunkte und andere Werte. Bei der ersten und einzigen Wertzuweisung wird ein Label neu definiert. Vorher ist dieses dem Assembler unbekannt.

Die Namen der Labels dürfen bis zu 63 Zeichen lang sein und aus Buchstaben und Zahlen bestehen. Das erste Zeichen muß jedoch immer ein Buchstabe sein. Um den Namen zu gliedern, dürfen der Apostroph (') und der Unterstrich (_) verwendet werden. Labels werden auf zwei verschiedene Arten Werte zugewiesen und damit auch definiert. Erstens, indem der Labelname dem Assemblerbefehl oder Pseudobefehl vorangestellt wird. Dadurch wird dem

Label die Adresse des aktuellen Programmpunkts, der beim Assemblieren errechnet wurde, zugewiesen:

```
LOOP    STA    BUF,X
```

Der zweite Weg wird durch den =-Befehl eröffnet. Hier wird dem Label direkt der Wert eines mathematischen Ausdrucks zugewiesen:

```
BUF     =  512+$1A0
```

Alle Zuweisungen dieser Art sollten am Programmanfang erfolgen. Eine mehrmalige Wertzuweisung ist mit dem "←"-Befehl möglich. Die Wertzuweisung erfolgt beim =-Befehl nur im 1. Paß und beim "←" -Befehl im ersten, und zweiten Paß. Um eine Parameterübergabe zwischen Programmen, die mit dem .FILE-Befehl verknüpft werden, oder zu einem Makro herzustellen, müssen globale Labels definiert werden. Dies geschieht bei der Labeldefinition durch Voranstellen eines Größerzeichens (>) vor den Namen.

```
>HELP   =      $800
>STOP   LDA    #123
```

Im ersten Paß muß bei der Verwendung eines Labels dieses nicht unbedingt definiert sein. Es kann damit z.B. bei einem Sprungbefehl eine Adresse durch ein Label vertreten werden, obwohl dieses durch den Assembler noch nicht erreicht wurde und somit auch das Label noch nicht definiert ist.

```
START  BNE  ENDE  ENDE noch nicht definiert
       INC  ADR
ENDE   RTS       Definition von ENDE
```

Im obigen Beispiel ist zu erkennen, daß das Label beim Aufruf im ersten Paß noch nicht definiert ist. Erst am Programmende wird diesem ein Wert zugewiesen und es damit definiert. Im zweiten Paß ist das Label bereits beim BNE-Befehl definiert, und der Objektcode kann ordnungsgemäß erzeugt werden.

Gekoppelte Labels

Es gibt eine besondere Art von Labelnamen, die in Assemblerprogrammen verwendet werden können. Dabei wird der Name eines Labels in Abhängigkeit von einem anderen bereits definierten Label festgesetzt. Hierzu ein Beispiel:

```
F        =    5
X        ←    110
ABC#F    =    $ABCD
TEST#X   =    123
X        ←    X+1
TEST#X   =    321
```

Der Assembler hängt bei Erreichen des Zeichens "#" an den Labelnamen den Zahlenwert des nachfolgenden Labels an. Zur Vereinfachung darf der Labelname des nachgestellten Labels nur ein Zeichen lang sein. Es sind also 26 verschiedene Anhängsel möglich. Hier die Ergebnisse der obigen Definitionen:

```
ABC0005     =   43981
TEST006E    =   123
TEST006F    =   321
```

Bei der Verwendung der Labels im Programm darf natürlich wieder die Schreibweise mit gekoppelten Labels verwendet werden. Hier eine mögliche Fortsetzung des Beispiels:

```
        LDA     TEST#X      X=111 => LDA 321
X       ←       X-1
        CMP     #TEST#X     X=110 => LDA #123
```

Die Labels finden Einsatz beim Aufbau komplexer Makro-Strukturen. Es ist damit z.B. möglich, Makros für strukturierte Programmierung zu bilden. Beispiele finden Sie in den mitgelieferten Programmen auf der EDASS-Diskette.

Verknüpfung von Programmen

Es gibt zwei Befehle, um Quelltexte beim Assemblieren zu verbinden. Der .END-Befehl am Programmende darf als Argument einen Programmnamen enthalten. Bei diesem wird die Assemblierung fortgesetzt. An das zweite Programm werden die gesamten Labels übergeben. Ein Labelname darf deshalb im aufgerufenen Programm nicht nochmals definiert werden. Auf diese Weise dürfen beliebig viele Programme verknüpft werden. Am Ende des ersten Paß ruft der Assembler automatisch wieder das erste Programm auf.

Eine zweite Art der Verknüpfung bietet der .FILE-Befehl. Hier wird das verbundene Programm wie ein Unterprogramm aufgerufen. Nach dem .FILE-Befehl verläßt der Assembler das Hauptprogramm, bearbeitet das aufgerufene Programm und kehrt zum Hauptprogramm zurück.

Der Objektcode sieht jedoch aus, als wäre er von einem fortlaufenden Programm erzeugt worden. Im Unterschied zum .END-Befehl werden die Labels nicht übergeben. Labelnamen dürfen also in beiden Programmen doppelt verwendet werden. Zur Parameterübergabe dienen globale Labels.

Diese werden bei der Definition, und nur dort, durch ein Größerzeichen (>") markiert. Die .FILE-Befehle dürfen geschachtelt werden. Zusammen mit den Makro-Aufrufen sind 255 Verknüpfungen erlaubt.

Makroassemblierung

Makros sind Befehlsfolgen, die innerhalb eines Programms mehrmals verwendet werden und zur Vereinfachung nur einmal eingegeben werden. Im Programm wird einfach, statt der einzelnen Befehle, der Name des Makros aufgeführt. Diese Befehlsfolgen werden bei jedem Aufruf neu in das Programm eingebaut. Ein solches Makro wird durch Parameter flexibel gemacht und kann damit speziellen Programmpositionen angepaßt werden.

Die Makro-Definition wird mit dem .MACRO-Befehl eingeleitet. Der vorangestellte Labelname ist der Name des Makros. Die nachfolgenden Befehle werden als Makro übernommen. Die Definition wird durch den Befehl .ENDM beendet.

Zur Parameterübergabe werden Labels verwendet. Deren Namen werden im .MACRO-Befehl als Argument angegeben. Die Definition und Wertzuweisung erfolgt beim Makroaufruf. Dieser erfolgt mit dem "/"-Befehl und nachstehendem Makronamen. Die zu übergebenden Parameter werden als Argumente des Befehls eingegeben. Hierzu ein Beispiel für Makros:

Definition:

```
        *=      $1400
SETXY   .MACRO  (XWERT,YWERT)
        LDX     #XWERT
        LDY     #YWERT
        .ENDM
```

Aufruf:

```
        ;
        /SETXY  (1,2)
        /SETXY  (3,100)
        RTS
        .END
```

Innerhalb des Makros dürfen neue Labels definiert werden. Diese können mit denen aus dem aufrufenden Programm identisch sein. Die Übergabe von Labelwerten, ohne Einsatz der Makroparameter, erfolgt über globale Labels.

Bedingte Assemblierung

Unter bedingter Assemblierung ist zu verstehen, daß ein Programmstück nur assembliert wird, wenn eine bestimmte Bedingung erfüllt ist. In einem Programm können z.B. Testroutinen eingebaut werden, die beim Assemblieren, abhängig vom Wert eines Labels, in den Objektcode übernommen oder weggelassen werden. Bei EDASS wird die bedingte Assemblierung durch einen . IF-Befehl (mit unterschiedlichen Bedingungen) eingeleitet und das Ende

durch einen .ENDIF-Befehl gekennzeichnet. Beim .IF-Befehl wird in einem mathematischen Ausdruck entschieden, ob das eingeschlossene Programmstück assembliert wird oder nicht.

Es können nun mehrere Programmblöcke aneinandergereiht oder auch ineinander geschachtelt werden. Auf diese Weise sind sehr komplexe Strukturen zu erreichen.

Vom Assembler erzeugte Dateien

Der Assembler kann die von ihm erzeugten Daten in Diskettendateien schreiben. Zur Unterscheidung werden die Dateinamen mit Kennbuchstaben versehen. Hier eine Zusammenstellung der gespeicherten Daten, der Befehle, die dieses erzeugen bzw. auslesen, und der zugehörigen Kennbuchstaben.

Assemblerlisting	(.LIST)	".L"
Objektcode	(.OBJ)	".O"
Labeltabelle	(.PUSH, .PULL)	".P"
Labelausgabe	(.LBL, .LBL$)	".T"
Fehlermeldungen	(.ERROR)	".E"

Die Pseudobefehle

Im folgenden werden die einzelnen Assembleranweisungen aufgezählt und beschrieben. Wie bei den EDASS-Befehlen aus Kapitel 15 gilt auch hier, daß Zahleingaben durch mathematische Ausdrücke ersetzt werden dürfen, ein Klammeraffe bzw. Paragraphenzeichen vor dem Dateinamen ein Überschreiben zur Folge hat und Programmnamen durch Jokerzeichen "?" und "*" abgekürzt werden können.

Neu ist folgendes: Wird im Programm ein neuer Ausgabebefehl (.LIST, .OBJ,...) gegeben, wird zuvor ein alter, geöffneter Kanal geschlossen, sozusagen der "OFF"-Befehl durchgeführt. Beim Bildschirm und Drucker bedeutet dies, daß ein Wagenrücklauf (CR=13) gesendet wird, und bei der Ausgabe auf Diskette, daß die geöffnete Datei geschlossen wird.

Für die Beschreibung der Befehle gilt auch hier, daß Angaben in kursiver Schrift nicht unbedingt erforderlich sind. Auf einer 1541/1571-Diskettenstation können maximal drei sequentielle Dateien geöffnet werden. Der Versuch, eine weitere Datei zu öffnen, verursacht einen Diskettenfehler. Um dies zu vermeiden, wird im folgenden bei jedem Befehl, der einen Diskettenzugriff einleitet, angegeben, in welchem Paß und wie lange er eine Datei geöffnet hält.

(Label) = (math. Ausdruck)

Mit diesem Befehl wird ein Label definiert und ein Wert zugewiesen. Existiert das Label bereits, führt eine nochmalige Definition zur Fehlermeldung:

```
?LABEL BEREITS DEFINIERT
```

Es ist nicht erlaubt, innerhalb des mathematischen Ausdrucks, dessen Wert dem Label zugewiesen werden soll, nicht definierte Labels zu verwenden.

(Label) ← (math. Ausdruck) oder (Label) < (math. Ausdruck)

Im Gegensatz zum =-Befehl ist dieser Befehl speziell zur nochmaligen Definition von Labels gedacht. Eine Umwandlung, z.B. eines lokalen Labels in ein globales Label, ist dabei nicht möglich. Die Wertzuweisung bei diesem Befehl erfolgt sowohl im ersten als auch im zweiten Paß.

Wird der Assembliervorgang durch ein Label gesteuert und wird dessen Wert mit dem ←-Befehl verändert, sollte auch die erstmalige Definition mit diesem Befehl erfolgen, damit in beiden Paß-Durchläufen das Label mit dem gleichen Wert startet. Hier ein kurzes Beispiel, das den Unterschied zwischen der Verwendung des =- und des ←-Befehls zeigt:

			1. Paß	2. Paß
ABC	=	100	100	110
ABC	←	ABC+10	110	120
ABC	←	100	100	100
ABC	←	ABC+10	110	110

Wie zu erkennen ist, können durch die Verwendung des =-Befehls die beiden Paß-Durchläufe beim Assemblieren unterschieden werden.

***= (math. Ausdruck)**

Mit dieser Anweisung wird der Programmzeiger des Assemblers auf einen neuen Wert gesetzt. Beim Start des Assemblers steht dieser Zähler automatisch auf der Adresse $1300, dem Beginn des freien Speicherbereichs im C128.

Wird der Objektcode auf die Diskette geschrieben, darf der Programmzeiger nicht auf eine niedrigere Adresse gesetzt und dort weiterer Objektcode erzeugt werden. Beispielsweise zur Reservierung von Speicher oder der Definition

von Labels darf dies dennoch geschehen. Auch in Zusammenhang mit dem .OFF-Befehl ist ein Zurücksetzen erlaubt. Die Fehlermeldung

```
?ZURUECKSETZEN NICHT ERLAUBT
```

weist Sie darauf hin, daß Sie einen der genannten Punkte nicht beachtet haben.

Durch die Befehlsfolge *=*+(Bereichsgröße) kann innerhalb des Maschinenprogramms ein Speicherbereich reserviert werden. Es wird dabei im mathematischen Ausdruck der alte Wert des Programmzeigers (* entspricht dem Wert des Programmzeigers) gelesen, um die Bereichsgröße erhöht und wieder dem Programmzeiger zugewiesen.

```
        *= *+10
TAB     *= *+$A0
```

/(Makroname) *(1. Parameter, 2. ...)*

Dieser Befehl ruft ein Makro mit dem im Befehl angegebenen Namen auf. Das Makro muß zuvor definiert worden sein, sonst erscheint die Fehlermeldung:

```
?MAKRO NICHT DEFINIERT
```

Die Parameter, die übergeben werden sollen, stehen in Klammern eingeschlossen und durch Kommata getrennt hinter dem Makronamen. Die Zahl der übergebenen Parameter muß nicht mit der vom Makro geforderten übereinstimmen. Sind zu wenige angegeben, wird den überschüssigen Labels des Makros der Wert Null zugewiesen. Sind zu viele Parameter eingegeben, werden die überzähligen mißachtet. Bei Makroaufrufen ohne Parameter entfallen alle Angaben hinter dem Makronamen.

Die bei der Parameterübergabe im Makro definierten Labels können nur während dieses einzelnen Aufrufs gelesen werden. Innerhalb des Makros sind weitere Labeldefinitionen erlaubt. Auch weitere Makroaufrufe sind möglich. Der Makroaufruf wird durch einen .ENDM- oder .END-Befehl beendet.

Tritt innerhalb des Makros beim Assemblieren ein Fehler auf, erscheint in der Fehlermeldung der Name des Makros und die Zeilennummer des definierenden Programms, in der die fehlerhafte Makrozeile zu finden ist. Daß es sich um einen Makronamen handelt, wird durch Voranstellen von "M:" kenntlich gemacht.

```
/ADD      (100,200)
/ADDXY
/TEST     (1,2,3,4,5,6,7,8,9)
```

.BANK (Wert von 0-15)

Dieser Befehl wählt für die Objektcodeausgabe in den Speicher (.OBJ M) die gewünschte Speicherbank des C128 aus. Der Befehl ist mit dem BASIC-Befehl BANK vergleichbar. Beim Start des Assemblers wird automatisch Bank 15 gewählt. Der Assembler verwendet zur Festsetzung der Bank dieselbe Speicherzelle wie BASIC. Die letzte Bank-Wahl des Assemblers kann deshalb bei den Befehlen PEEK, POKE, SYS sowie !GO, !REASS und !BYTE weiterverwendet werden.

.BYTE (8-Bit-Zahl),*(8-Bit-Zahl)*,...

Der .BYTE-Befehl dient zum Einbau von Zahlenwerten mit der Länge eines Bytes in den Objektcode. Die einzelnen Zahlenwerte werden durch Kommata (,) voneinander getrennt. Ist ein Zahlenwert größer als 255, d.h. länger als ein Byte, erscheint die Fehlermeldung:

```
?UNERLAUBTES ARGUMENT

.BYTE    123
.BYTE    23,"A",%101*43
```

.DBYTE (16-Bit-Zahl),*(16-Bit-Zahl)*,...

Mit dem .DBYTE-Befehl fügen Sie eine Folge von 16-Bit Zahlen in den Objektcode ein. Es wird zuerst das HI-Byte und dann das LO-Byte der Zahl eingebaut, also genau entgegengesetzt der Adreßspeicherung des 6502- bzw. 8502-Prozessors. Wie beim .BYTE-Befehl werden auch hier die einzelnen Werte durch Kommata getrennt.

```
.DBYTE $ABCD
.DBYTE $DF45,45,"A"*"N"
```

.DISK "(Befehlsstring)",*(Geräteadr.)*

Dieser Befehl dient zum Übersenden eines Diskettenkommandos. Tritt ein Diskettenfehler nach Abschluß des Befehls ein, wird dieser ausgegeben. Der Befehl wird im ersten und zweiten Paß ausgeführt. Er belegt keinen Dateikanal.

Benutzer des C128, die sich nur mit den Diskettenbefehlen des BASIC auskennen, sollten im Handbuch des Diskettenlaufwerks nachschlagen, um zu erfahren, wie die verschiedenen Kommandos lauten

```
.DISK "I0"              Laufwerk initialisieren
.DISK "S0:TEST"         Datei Test löschen
```

.END ***"(Programmname)"***

Der .END-Befehl markiert das Ende eines Programmtextes. Dies kann das Ende eines Hauptprogramms oder das Ende eines mit .FILE verbundenen Programms sein. Der Assembler erkennt auch selbständig das Ende des Quelltextes. Die .END-Anweisung sollte der Übersichlichkeit halber immer vorhanden sein.

Wird als Argument des Befehls ein Programmname angegeben, setzt der Assembler seine Arbeit mit diesem Programm fort. Alle Labels werden an dieses übergeben. Der Programmtext kann sich im Computerspeicher oder auf Diskette befinden. Mit diesem Befehl dürfen beliebig viele Programme verknüpft werden. Bei Erreichen eines .END-Befehls ohne Argument wird der erste Paß beendet, und der Assembler kehrt automatisch zum ersten Programm der Aufruffolge zurück.

.ENDM

Nach einem .MACRO-Befehl beendet der .ENDM-Befehl die Makro-Definition. Die zwischen beiden Befehlen eingeschlossenen Befehle werden als Makro getrennt gespeichert. Beim Aufruf eines Makros signalisiert der .ENDM-Befehl das Ende des Makros und gibt damit das Kommando zur Rückkehr ins aufrufende Programm. Der .ENDM-Befehl darf nur in Zusammenhang mit Makros verwendet werden. Taucht er an falscher Stelle auf, erscheint die Fehlermeldung:

```
?KEIN MAKRO AUFGERUFEN
```

.ERROR (Geräteadr.),*(Sekundäradr.)*

Tritt während des Assemblierens ein Fehler auf, wird die Fehlermeldung, der Name des Programms und die Zeilennummer, in der er aufgetreten ist, ausgegeben. Der Assembler bricht danach seine Arbeit ab. Der .ERROR-Befehl verhindert den Abbruch. Die Meldungen werden auf das im Befehl angegebene Gerät geleitet.

Folgende Fehler können nicht unterdrückt werden, da bei ihnen eine Fortsetzung nicht möglich ist oder sie einen Absturz des Computers verursachen könnten:

```
?GERAET NICHT ANGESCHLOSSEN
?ZU VIELE VERKETTUNGEN
?KEIN PROGRAMMSPEICHER MEHR
?KEIN LABELSPEICHER MEHR
?MAKRO IN MAKRO DEFINIERT
?KEIN MAKRO AUFGERUFEN
?ZURUECKSETZEN NICHT ERLAUBT
?ZU VIELE OFFENE DATEIEN
```

Diskettenfehler

Der Objektcode einer fehlerhaften Zeile wird nicht erzeugt. Deshalb sollte ein Programm, bei dem Fehler aufgetreten sind, nicht gestartet werden.

Wurde der Assembler vom Editor aus aufgerufen und werden die Fehlermeldungen auf den Bildschirm geleitet, erscheinen die Texte in der Statuszeile. Damit die Meldungen gelesen werden können, wartet EDASS, bis eine Taste gedrückt wurde.

.ERROR "(Dateiname)"*,(Geräteadr.)*

Dieser Befehl hat dieselbe Wirkung wie der vorige .ERROR-Befehl, jedoch werden diesmal die Fehlermeldungen auf Diskette gespeichert. Um die Datei von anderen zu unterscheiden, wird an den Namen ".E" angehängt. Der Befehl wird im ersten Paß aufgerufen und öffnet dort eine Datei. Diese bleibt dann bis zum Ende des zweiten Paß geöffnet, es sei denn, der .ERROR O-Befehl wird durchgeführt.

.ERROR O

Das "O" steht bei diesem Befehl für "OFF". Damit wird ein vorher gegebener .ERROR-Befehl deaktiviert und der Abbruch des Assemblierens wieder eingeschaltet. Ein auf Diskette geöffnetes ERROR-File wird dabei geschlossen. Die Fehlermeldungen erscheinen wieder in normaler Art und Weise.

.FAST

Wie der gleichlautende BASIC-Befehl schaltet auch dieser Befehl den C128 auf den schnelleren Arbeitsmodus. Das Umschalten erfolgt während des Assemblierens. War beim Start des Assemblers der SLOW-Modus aktiv, schaltet der Assembler am Ende seiner Arbeit automatisch auf diesen Modus zurück. Mit dieser Einrichtung kann im 40-Zeichen-Modus eine schnellere

Assemblierung herbeigeführt werden. Ausgaben auf den Bildschirm sind natürlich dann nicht zu sehen.

.FF

Bei formatierter Ausgabe bewirken Sie mit diesem Befehl einen Seitenvorschub im Listing. Es wird dabei das Papier bis zu der im .FORMAT-Befehl angegebenen Papiergrenze transportiert und, wenn gewünscht, eine Seitennummer ausgegeben. Am Ende des Assembliervorgangs wird automatisch ein Vorschub durchgeführt.

.FILE "(Programmname)",*(Geräteadr.)*

Dieser Befehl dient zum Verknüpfen von Programmen. Das angesprochene Programm kann im Speicher oder auf Diskette stehen. Nach Aufruf des Befehls setzt der Assembler seine Arbeit im neuen Programm fort. Dieses wird wie ein Unterprogramm angesprochen. Hat er dessen Ende erreicht, kehrt der Assembler wieder in das Hauptprogramm zurück.

Ein solches Unterprogramm kann seinerseits wieder ein Programm mit dem .FILE-Befehl aufrufen. Der erzeugte Objektcode sieht aus, als handle es sich um ein fortlaufendes Programm. Die Labels der verknüpften Programme werden getrennt. Es dürfen also gleiche Namen vorkommen. Zur Übergabe von Labelwerten zwischen den Programmen dienen globale Labels.

Der Befehl belegt im ersten und zweiten Paß während der Abarbeitung des aufgerufenen Programms einen Dateikanal. Insgesamt dürfen zusammen 255 .FILE und .MACRO-Aufrufe erfolgen.

.FORMAT (Papierlänge),(Textlänge),(oberer Rand),
(linker Rand),(rechter Rand)

Dieser Befehl schaltet die formatierte Ausgabe ein. Dies bedeutet, daß das Assemblerlisting und alle anderen Ausgaben auf das gleiche Gerät jetzt seitenweise mit einem linken und rechten Rand ausgegeben werden.

Der Parameter Papierlänge gibt an, wie lang das Papier ist, und der Parameter Textlänge, wie viele Zeilen davon bedruckt werden sollen. Mit dem nächsten Wert legen Sie fest, wie viele Zeilen am oberen Rand der Seite freigelassen werden sollen. Die letzten beiden Parameter schließlich bestimmen den linken und rechten Rand des Textes auf dem Papier.

Nach jeder Seite wartet der Computer auf einen Tastendruck. Sie haben damit die Möglichkeit, Einzelblätter zu verarbeiten.

Die Formatierung wirkt auf alle Ausgaben, die auf das mit dem .LIST-Befehl angesprochende Gerät laufen. Wird z.B. die Ausgabe der Fehlermeldungen mit dem .ERROR-Befehl auf dasselbe Gerät geleitet, werden auch die Fehlermeldungen formatiert.

.FORMAT O

Mit diesem Befehl schalten Sie die Formatierung wieder ab. Das Listing erscheint wieder in loser Form.

.IF (math. Ausdruck) .....ENDIF

Die Kombination dieser beiden Pseudo-Befehle ermöglicht bedingte Assemblierung, d.h. Programmstücke werden nur assembliert, wenn eine bestimmte Bedingung erfüllt ist. Bei EDASS wird das eingeschlossene Programmstück nur assembliert, wenn der mathematische Ausdruck eines vorangehenden .IF-Befehls einen Wert ungleich null hat. Ist der Wert des Ausdrucks null, wird zum .ENDIF-Befehl gesprungen und das dazwischenliegende Programmstück nicht assembliert.

Fallen das Ende des Programmstücks und das Ende des Programms zusammen, kann der .ENDIF-Befehl weggelassen werden. Der Programmabschnitt wird dann durch das Programmende oder einen .END-Befehl, bzw. bei Makros einen .ENDM-Befehl begrenzt.

```
.IF     1+2                 ;WERT UNGLEICH NULL
LDA     #0                  ;=> PROGRAMMBLOCK WIRD
STA     53280               ;ASSEMBLIERT
.ENDIF
NOP
.IF     1-1                 ;WERT UNGLEICH NULL
LDA     #0                  ;=> PROGRAMMBLOCK WIRD
STA     53281               ;NICHT ASSEMBLIERT
.ENDIF
RTS
```

Es gibt noch weitere .IF-Befehle, die auf andere Ergebnisse des mathematischen Ausdrucks reagieren:

```
.IFNE    Wert ungleich null (wie .IF)
.IFEQ    Wert gleich null
.IFPL    Wert positiv (Zweierkomplement-Arithmetik)
.IFMI    Wert negativ (Zweierkomplement-Arithmetik)
```

Es dürfen beliebig viele .IF-.ENDIF Strukturen mit unterschiedlichen Bedingungen ineinander geschachtelt werden.

.LBL (Geräteadr. 3 – 7),*(Sekundäradr.)*

Mit diesem Befehl ist es möglich, alle Labels, die definiert und von diesem Programmpunkt aus lesbar sind, auszugeben. Zu diesen gehören alle Labels des augenblicklich bearbeiteten Einzelprogramms (Hauptprogramm, .FILE-Verknüpfung) bzw. Makros und alle definierten globalen Labels. Die einzelnen Labels werden in der Reihenfolge, in der sie definiert wurden, ausgegeben. Es erscheint der Name, ein "="-Symbol und dann der Wert des Labels in dezimaler Darstellung. Handelt es sich um ein globales Label, wird vor dem Namen ein Größerzeichen (>) gesetzt. Die Ausgabe findet im zweiten Paß statt, d.h. eine Labelliste steht innerhalb eines evtl. Programmlistings.

.LBL "(Dateiname)",*(Geräteadr.)*

Dieser Befehl hat dasselbe Ergebnis wie der normale .LBL-Befehl, jedoch wird als Ausgabeort eine Diskette gewählt. Zur Unterscheidung von anderen Dateinamen wird an das Ende des Namens ".T" gehängt. Eine Datei wird im 2. Paß geöffnet und nach Abschluß des Befehls wieder geschlossen. Der .LBL-Befehl darf nicht mit .PUSH verwechselt werden. Die mit dem .LBL-Befehl gespeicherten Labels können nur zur späteren Ausgabe einer Labeltabelle verwendet werden.

.LBL$ (Geräteadr. 3 – 7),*(Sekundäradr.)*

Gegenüber dem .LBL-Befehl gibt dieser Befehl die Werte der Labels in hexadezimalen Zahlen aus.

.LBL$ "(Dateiname)",*(Geräteadr.)*

Dieser Befehl schreibt im Unterschied zum entsprechenden .LBL-Befehl den Wert eines Labels im Hexadezimalsystem auf die Diskette.

.LIST (Geräteadr. 3 – 7),*(Sekundäradr.)*

Dieser Befehl veranlaßt den Assembler, ein Listing auszugeben. In einer Zeile des Listings erscheint zuerst der Wert des Programmzählers im HEX-Format, dann der Objektcode ebenfalls im HEX-Format, die Zeilennummer, unter der

die Programmzeile wieder zu finden ist, und zum Schluß der eigentliche Programmtext der Zeile. Die Ausgabe des Programmzählers ist um die Angabe der aktuellen Banknummer erweitert. Diese erscheint vor der Hexadezimalzahl, wodurch der Programmzähler fünf Ziffern umfaßt. Beim Aufruf eines Makros wird die Ausgabe der Makrozeilen unterdrückt. Es erscheint nur der Aufruf. Bei intensiver Verwendung von Makros wird das Listing dadurch übersichtlicher und überschaubarer. Ausgaben aus dem Makro heraus, z.B. mit dem .PRINT-Befehl, sind weiter möglich. Die Assemblerschlußmeldung ("ENDE DER ASSEMBLIERUNG!", "PASS-DIFFERENZ !") wird bei der Ausgabe des Listings anstatt auf den Bildschirm ans Ende des Listings geschrieben.

LIST "(Dateiname)",*(Geräteadr.)*

Durch diesen Befehl kann eine Diskette als Ausgabeort für das Listing gewählt werden. An den Dateinamen wird zur Unterscheidung automatisch ".L" angehängt. Der Befehl öffnet im zweiten Paß eine Datei, die bis zum Programmende oder einem neuen .LIST-Befehl geöffnet bleibt.

.LIST O

Dieser Befehl schaltet die Ausgabe des Assemblerlistings wieder ab. Eine mit dem .LIST-Befehl geöffnete Datei wird dabei geschlossen.

.LISTM (Geräteadr. 3 – 7),*(Sekundäradr.)*

Im Gegensatz zum .LIST-Befehl erscheint bei diesem Befehl bei einem Makroaufruf auch das Assemblerlisting der Makrozeilen. Bei der Fehlersuche innerhalb eines Makros kann dies sehr nützlich sein.

.LISTM "(Dateiname)",*(Geräteadr.)*

Dieser Befehl leitet das Assemblerlisting in eine Datei. Wie beim vorangegangenen Befehl wird auch hier das Assemblerlisting der Makros erzeugt.

.LISTM O

Dieser Befehl schaltet die Ausgabe eines Assemblerlistings wieder ab. Beim Abschalten der Ausgabe ist es egal, ob der Befehl .LIST O oder der Befehl .LISTM O eingesetzt wird.

(Makroname) .MACRO *(Labelnamen,...)*

Dieser Befehl leitet eine Makro-Definition ein. Der Name des Makros wird wie ein Label dem Befehl vorangestellt. Es gibt dabei keinen Unterschied zwischen lokalen und globalen Labelnamen. Ein Makro ist grundsätzlich global.

Zur Parameterübergabe werden Labels definiert und Werte zugewiesen. Die Namen dieser Labels stehen durch Klammern eingeschlossen hinter dem Befehl. Bis zu zehn Parameter, also zehn Labelnamen, dürfen verwendet werden.

Die einzelnen Namen werden innerhalb der Klammern durch Kommata getrennt. Es sind auch Makro-Definitionen ohne Parameterübergabe erlaubt. In diesem Fall werden einfach die Labelnamen samt Klammern weggelassen.

Beim späteren Aufruf der Makros werden die übergebenen Parameter den Labels zugewiesen. Die Labels selbst können nur innerhalb dieses Makroaufrufs gelesen werden. Sie können nicht über den Aufruf hinweg gerettet werden. Hierzu ein paar Beispiele für .MACRO-Befehle:

```
ADDXY    .MACRO
ADD      .MACRO          (SUM1,SUM2)
TEST     .MACRO          (PRUEF,FEHLER,ENDE)
```

Alle nachfolgenden Programmzeilen werden als Makro erfaßt. Die Definition endet mit einem .ENDM-, .END-Befehl oder dem Programmende. Steht das definierte Makro im Speicher, merkt sich der Assembler nur die Startadresse der Programmstelle. Wird das Programm jedoch von Diskette assembliert, also von dort eingelesen, erstellt der Assembler vom Makro im Speicher eine Kopie.

Beim späteren Aufruf muß jetzt nicht die Datei gelesen werden, sondern das Makro kann aus dem Speicher entnommen werden. Die Makro-Definitionen teilen sich den Speicher mit dem Programmtext. Bei einer großen Zahl von Makros, die von der Diskette eingelesen werden, sollte deshalb der Programmspeicher möglichst leer sein. Reicht der Programmspeicher nicht mehr aus, erscheint die Meldung:

```
?KEIN PROGRAMMSPEICHER MEHR
```

Die Definition eines Makros innerhalb eines anderen Makros ist nicht erlaubt. Es erscheint der Fehler:

```
?MAKRO IN MAKRO DEFINIERT
```

.OBJ M

Mit diesem Befehl bewirken Sie, daß der Objektcode in den Computerspeicher geschrieben wird. Der Assembler gibt ohne einen derartigen Befehl den Objektcode nicht aus.

.OBJ D *(Geräteadr.)*

Mit diesem Befehl veranlassen Sie den Assembler, den Objektcode in den Diskettenspeicher zu schreiben. Ein dort stehendes Programm kann nach dem Assemblieren mit dem Befehl !GO D gestartet werden.

.OBJ (Geräteadr. 3 – 7),*(Sekundäradr.)*

Dieser Befehl bewirkt, daß der Objektcode auf dem Bildschirm oder einem Drucker ausgegeben wird. Die Ausgabe ist mit dem HEX-Dump eines Monitors zu vergleichen. In einer Zeile erscheinen zuerst der Programmzähler und dann 16 Bytes Objektcode. Alle Werte werden im HEX-Format ausgegeben. Ein Umsetzen des Programmzeigers wird in der Ausgabe durch einen Absatz gekennzeichnet.

.OBJ "(Dateiname)",*(Geräteadr.)*

Mit diesem Befehl leiten Sie den Objektcode in eine Datei auf der Diskette um. An den Dateinamen wird zur Kennzeichnung ".O" angehängt. Sobald der Befehl aufgerufen wird, schreibt der Assembler den aktuellen Stand des Programmzeigers als Startadresse in die Datei. Auf diese Weise kann der gespeicherte Objektcode mit dem BASIC-Befehl BLOAD wieder eingelesen werden.

Es sollte wegen dieser Vorgehensweise darauf geachtet werden, daß zuerst der Programmzeiger gesetzt wird (*=-Befehl) und dann der .OBJ-Befehl gegeben wird. Nur damit ist gewährleistet, daß in die Datei wirklich die Startadresse des nachfolgenden Programms eingetragen wird.

Während der Objektcode auf Diskette geschrieben wird, darf der Programmzeiger nicht auf eine niedrigere Adresse gesetzt und dort Objektcode erzeugt werden. Ein bloßes Umsetzen zum Definieren von Labels oder ein Umsetzen in Verbindung mit dem .OFF-Befehl ist erlaubt. Wird der Programmzeiger zu einer höheren Adresse versetzt, füllt der Assembler die entstehende Lücke mit Null-Bytes aus.

Die Ausgabedatei wird vom Befehl im 2. Paß geöffnet und bleibt bis zum Programmende oder einem neuen .OBJ-Befehl offen.

.OBJ (Memoryangabe),(Peripheriegerät)

Dieser Befehl ist eine Kombination aus den vorangegangenen .OBJ-Befehlen. Es ist damit möglich, den Objektcode in den Computer- oder Diskettenspeicher zu schreiben und ihn gleichzeitig auf dem Bildschirm, Drucker oder der Diskette auszugeben. Im Befehl muß zuerst die Speicherangabe und dann die Angabe eines Peripheriegeräts erfolgen. Beide Angaben werden durch ein Komma voneinander getrennt.

```
.OBJ M,4                    ;COMPUTER UND DRUCKER
.OBJ D,3                    ;DISKSPEICHER UND BILDSCHIRM
.OBJ M,"TEST"               ;COMPUTERSPEICHER UND DISKFILE
```

.OBJ O

Mit diesem Befehl schalten Sie die Ausgabe des Objektcodes wieder aus. Eine geöffnete Datei für die Ausgabe des Objektcodes wird dabei geschlossen.

.OFF (16-Bit-Zahl)

Der Assembler verwendet einen Programmzeiger, um den Objektcode zu errechnen. Zum Festsetzen der Adresse beim Schreiben des Objektcodes wird ein zweiter Zeiger eingesetzt. Normalerweise laufen beide Zeiger parallel. Durch den .OFF-Befehl kann ein Offset (Differenz) des Schreibzeigers zum Programmzeiger festgesetzt werden. Der Objektcode kann dann z.B. ab der Adresse $1300 in den Speicher geschrieben werden, obwohl der Assembler für die Berechnung z.B. die Adresse $1C00 verwendet. Wird der Objektcode in eine Datei geschrieben, kann es beim Zurücksetzen des Programmzeigers zu einer Fehlermeldung kommen.

Den Ausschlag für die Fehlermeldung gibt jedoch nicht der Programmzeiger, sondern der Schreibzeiger. Wird dieser parallel zum Programmzeiger zurückgesetzt und wird dann Objektcode erzeugt, erscheint eine Fehlermeldung. Durch den .OFF-Befehl kann ein Umsetzen des Programmzeigers beim Schreibzeiger wieder ausgeglichen werden.

Zuerst wird der Schreibzeiger zusammen mit dem Programmzeiger durch den *=-Befehl zurückgesetzt und dann wieder mit dem .OFF-Befehl heraufgesetzt. Hierzu ein kurzes Beispielprogramm:

```
*=    $1300          ;PROGRAMMZEIGER STARTEN
LDA   #0
*=    $1102          ;PROGRAMMZEIGER ZURÜCKSETZEN
.OFF  $200           ;SCHREIBZEIGER WIEDER HOCHSETZEN
LDX   #1
RTS
.END
```

Wenn Sie dieses Programm mit dem Befehl !GO$1300 starten, kehrt das Programm wieder ordnungsgemäß mit RTS zurück. Das Umsetzen der Programmzeiger und damit auch des Schreibzeigers wurde für diesen wieder durch den .OFF-Befehl ausgeglichen.

Hierzu Anwendungen für den Befehl: Größere Maschinenprogramme werden meist als BASIC-Programme geladen. Ein ebenfalls in Maschinensprache geschriebener Lader kopiert den Objektcode an seine richtige Speicherlage. Sollen Lader und Maschinenprogramm als fertiges Programm vorliegen, müßten beide Teile getrennt assembliert und mit einem Monitor zusammengeflickt werden. Mit dem .OFF-Befehl können jedoch beide Teile gleich beim Assemblieren verbunden werden. Hier das rohe Gerüst dieses Vorgangs:

```
STARTLADER  =       $1C00
STARTPRG    =       $1300
            *=      STARTLADER
            .FILE   "LADER"
PRGLOAD     =       *
            *=      STARTPRG
            .OFF    PRGLOAD-STARTPRG
            .FILE   "PRG"
            .END
```

Der Lader nimmt an, daß direkt in seinem Anschluß der geladene Objektcode liegt. Dieser soll von ihm kopiert werden. Die Speicheradresse wird im Label PRGLOAD festgehalten.

Der Programmzeiger wird auf die Startadresse des Programms gesetzt, der Schreibzeiger aber wieder mit .OFF zurück auf die Position nach dem Lader. Als Ergebnis erhält man das fertige Programm. Da hierbei auch kein Zurücksetzen des Schreibzeigers stattfindet, kann der Objektcode auch sofort in eine Datei geschrieben werden.

.PAGE (Seitennummer)

Dieser Befehl veranlaßt den Assembler, bei formatierter Ausgabe auf jede Seite eine Seitennummer zu drucken. Diese erscheint in der Mitte des unteren Randes. Die Zählung beginnt bei der im Befehl angegebenen Nummer.

.PAGE O

Dieser Befehl schaltet die Ausgabe der Seitennummer wieder ab.

.PRINT "(Drucktext)",(Zahlenwerte),...

Dieser Befehl ist mit dem BASIC-Befehl PRINT zu vergleichen. Es können Texte und Zahlenwerte ins Assemblerlisting gedruckt werden. Beispielsweise können dem Programmierer die Werte wichtiger Labels oder Berechnungen gezeigt werden. Texte und Zahlenwerte können beliebig gemischt werden. Die einzelnen Angaben werden durch Kommata getrennt. Zahlen erscheinen im Dezimalsystem. Am Ende des Befehls sendet der Assembler automatisch einen Wagenrücklauf. Der .PRINT-Befehl selbst erscheint im Listing nicht.

```
.PRINT "STARTADRESSE:",START
.PRINT "ENDEADRESSE :",ENDE
.PRINT ENDE-START,"(LAENGE)"
```

.PRINT$ "(Drucktext)",(Zahlenwerte),...

Dieser Befehl gibt wie der zu vorige .PRINT-Befehl Texte und Zahlenwerte im Listing aus. Die Zahlenwerte erscheinen jedoch in hexadezimaler Form.

.PULL "(Dateiname)",(Geräteadr.)

Mit diesem Befehl wird eine mit dem .PUSH-Befehl auf Diskette gespeicherte Labeltabelle wieder eingelesen. Unbekannte Labels werden dabei neu definiert, bereits bestehende erhalten einen neuen Wert, und normale Labels können zu globalen Labels werden. Der Befehl öffnet im 1. Paß eine Datei und schließt diese wieder am Ende des Befehls.

.PUSH "(Dateiname)",(Geräteadr.)

Durch diesen Befehl können alle bisher definierten und vom Programmpunkt lesbaren Labels auf Diskette gespeichert werden. Zu diesen zählen alle im Einzelprogramm (Hauptprogramm, .FILE-Verknüpfung) bzw. Makro definierten Labels und alle bisher definierten globalen Labels.

An den Dateinamen wird zur Kennzeichnung ".P" angehängt. Der Befehl öffnet im ersten Paß eine Datei und schließt diese wieder nach Abschluß der Programmzeile.

.SLOW

Dieser Befehl ist das Gegenstück zum .FAST-Befehl. Der C128 wird wieder auf die langsame Arbeitsgeschwindigkeit geschaltet. Im 40-Zeichen-Modus kann der Befehl z.B. nach .FAST benutzt werden, um den Bildschirm wieder einzuschalten, damit Ausgaben des Assemblers sichtbar werden.

.TEXT "(String)",(8-Bit-Zahl),...

Mit diesem Befehl fügen Sie die ASCII-Werte der Zeichen eines Strings in den Objektcode ein. Außer Strings kann dieser Befehl auch 8-Bit-Zahlen, wie der .BYTE-Befehl, einbauen. Alle Elemente müssen durch Kommata getrennt werden.

```
.TEXT "ABCD"
.TEXT 0
.TEXT "UP",0,"DOWN",0
```

.TITLE (Zeilenposition),"(Überschrift)"

Mit diesem Befehl können Sie bei formatierter Ausgabe an den Kopf jeder Seite eine Überschrift setzen. Die Überschrift wird dazu in den freien oberen Rand des Papiers gedruckt. In welcher Zeile die Ausgabe erfolgt, geben Sie im Befehl mit der Zeilenposition an.

.TITLE 0

Dieses Kommando schaltet die Ausgabe einer Überschrift wieder ab.

.VIDEO "(String)",(8-Bit-Zahl),...

Wie der .TEXT-Befehl fügt auch diese Pseudoanweisung Strings und 8-Bit-Zahlen in den Objektcode ein. Hier werden jedoch die einzelnen Zeichen und Zahlen, die den ASCII-Code darstellen, in den CBM-eigenen Bildschirmcode umgerechnet und in dieser Form eingefügt. Ein solches Speicherformat ist z.B. für Meldungen, die dirket in den Bildschirmspeicher geschrieben werden, sehr nützlich. Die einzelnen Zeichen können direkt in das Programm eingetragen werden und müssen nicht von "Hand" umgerechnet werden.

```
.VIDEO "MELDUNG"
.VIDEO 1,2,3
.VIDEO "UP",$FF,"DOWN",$FF
```

.WORD (16-Bit-Zahl),(16-Bit-Zahl)

Dieser Befehl baut 16-Bit-Zahlen in den Objektcode ein. Im Gegensatz zum .DBYTE-Befehl wird zuerst das LO-Byte und dann das HI-Byte der Zahl eingefügt. Im selben Speicherformat verarbeitet auch der 6502- bzw. 8502-Prozessor 16-Bit-Zahlen. Der .WORD-Befehl kann aus diesem Grund z.B. sehr gut für Sprungtabellen verwendet werden.

```
.WORD $FEDC
.WORD 1234,$456,%1010*4321
.WORD START,ENDE,MITTE
```

.WRITE "(Drucktext)",(8-Bit-Zahl),...

Dieser Befehl dient der Gestaltung des Listings bzw. dem Senden von Steuercodes an den Drucker. Es können dazu mehrere Strings mit Text oder auch 8-Bit-Zahlen als Steuercodes ausgegeben werden. Die einzelnen Elemente werden durch Kommata getrennt. Am Ende des Befehls sendet der Assembler automatisch einen Wagenrücklauf (CR=13). Der .WRITE-Befehl selbst erscheint im Listing nicht.

```
.WRITE "ANFANG"
.WRITE 27,"!",4
.WRITE "START",13,"ENDE"
```

Kapitel 17

EDASS-Rechenfunktionen

EDASS ist mit einer eigenen Routine ausgerüstet, um mathematische Ausdrücke auszuwerten. Diese werden von links nach rechts abgearbeitet. Dabei können beliebig viele Klammerebenen verwendet werden.

Zahlenwerte

Im folgenden werden alle Möglichkeiten aufgezählt, Zahlenwerte in einen mathematischen Ausdruck einzubauen, um diese dann durch mathematische Operationen zu verknüpfen.

Werden Ziffern oder andere Zeichen verlangt, werden in der Beschreibung stellvertretend Nullen (0) angegeben. Die Länge einer Zahl kann beliebig gewählt werden. Führungsnullen sind nicht notwendig.

00000	Dezimalzahl (Ziffern 0-9)
$0000	HEX-Zahl (Ziffern 0-9 und A-F)
@0000/§0000	Oktalzahl (Ziffern 0-7)
%0000	Binärzahl (Ziffern 0-1)
"(String)"	ASCII-Wert des ersten Zeichens
*	Programmzeiger des Assemblers
&	Zahl der noch freien Programmbytes
-...	negativer Wert einer Zahl (-$ABCD,-"A",-*,...)
>...	HI-Byte des nachfolgenden Ausdrucks
<...	LO-Byte des nachfolgenden Ausdrucks
[...]	Wert einer Speicherzelle
(...)	Ausdruck in Klammern

Mathematische Verknüpfungen

Mit den folgenden Symbolen können die oben aufgeführten Zahlenwerte verknüpft werden:

+	addieren
–	subtrahieren
*	multiplizieren
/	dividieren

MOD	Restbyte einer Division
AND	logisch UND-verknüpfen
OR	logisch ODER-verknüpfen
XOR	logisch EXKLUSIV-ODER-verknüpfen

Bei der Verwendung der letzten vier Befehle im Zusammenhang mit Labels muß zwischen dem Namen und der Verknüpfung eine Leerstelle eingegeben werden. Nur so kann EDASS beide Teile trennen.

Kapitel 18

Fehlermeldungen und andere Nachrichten

EDASS verwendet deutsche Fehlermeldungen, die sich weitgehend selbst erklären. Der Assembler fügt den Meldungen noch den Ort des Auftretens bei. Dies sieht dann wie folgt aus:

```
?FEHLER IM BEFEHLSFORMAT          KREIS/0002
?ZU GROSSER RELATIVER SPRUNG      STERNE/0110
?UNERLAUBTES ARGUMENT             M:MAKRO/0213
```

Es wird der Name des Programms und die Nummer der fehlerhaften Zeile angegeben. Das letzte Beispiel zeigt eine Fehlermeldung bei der Assemblierung eines Makros. Dies wird durch "M:" gekennzeichnet. Nachfolgend steht der Name des Makros und die Zeilennummer im definierten Programm, in der die fehlerhafte Zeile wiederzufinden ist. Im folgenden werden alle Fehlermeldungen, ihre Ursache und Möglichkeiten, sie zu beheben, aufgezählt. Der Vollständigkeit halber sind auch alle sonstigen Meldungen von EDASS aufgeführt.

?ABGEBROCHEN

Diese Meldung erscheint, sobald eine Routine mit der RUN/STOP-Taste unterbrochen wurde.

?DIVISION DURCH NULL

Dieser Fehler wird angezeigt, sobald eine Division durch null versucht wird. Um den Fehler zu beheben, prüfen Sie am besten nach, ob im fehlerhaften Ausdruck alle verwendeten Labels den richtigen Wert haben.

?EINFUEGEN NICHT MOEGLICH

Diese Meldung wird vom INSERT-Befehl ausgegeben und teilt mit, daß sich die Eingabezeile im einzufügenden Bereich befindet. Dieselbe Meldung erscheint auch beim Versuch, ein ganzes Programm in sich selbst einzufügen.

ENDE DER ASSEMBLIERUNG!

Dies ist die Schlußmeldung des Assemblers. Zusätzlich erscheint die Endadresse des Programms. Auch die letzte Banknummer wird ausgegeben.

ERSETZEN?

Wird beim FIND-Befehl neben dem Suchstring auch ein Ersatztext eingegeben, fragt der Befehl bei einer gefundenen Programmstelle, ob der Ersatzstring eingesetzt werden soll. Die Frage wird durch die Tasten <J> oder <RETURN> mit "Ja" beantwortet, alle anderen Tasten verneinen sie.

?FEHLENDES LABEL

Der Assembler gibt diese Meldung aus, wenn beim Pseudo-Befehl =, ← , < oder .MACRO der Labelname am Zeilenanfang fehlt.

?FEHLER IM BEFEHLSFORMAT

Diese Meldung wird bei einem Fehler in dem vom Befehl verlangten Eingabeformat ausgegeben. Dies können fehlende Kommata, nicht existierende mathematische Verknüpfungen oder andere Dinge sein. Um den Fehler zu beheben, überprüfen Sie nochmals Ihren eingetippten Befehl. Der Fehler kann mit dem von BASIC her bekannten "SYNTAX ERROR" verglichen werden.

?FEHLERHAFTE BLOCKMARKEN

Diese Fehlermeldung erscheint im Editor. Sie gibt an, daß versucht wurde, die Blockbegrenzungen in einem Befehl zu verwenden, obwohl eine Blockmarke nicht gesetzt oder die Blockendemarke kleiner als die Blockanfangsmarke ist. Der Fehler entsteht nicht mehr, sobald die Marken richtig gesetzt sind.

?FEHLERHAFTES LABEL

Bei der Definition oder beim Aufruf eines Labels wurden nicht erlaubte Zeichen verwendet. Ein Labelname darf nur aus Buchstaben und Zahlen bestehen.

Zur Gliederung des Namen sind der Apostroph (') und der Unterstrich (_) erlaubt. Wird ein gekoppeltes Label verwendet (ABC#A) und ist das angehängte Label noch nicht definiert, erscheint diese Meldung auch.

?FUNKTIONSUNFAEHIGER SPRUNG

Alle Prozessoren der 65xx-Serie und auch der 8502 besitzen einen Herstellungsfehler. Dadurch sind alle indirekten Sprünge der Form JMP ($xxFF) funktionsunfähig, sobald das LO-Byte den Wert $FF hat. Die Fehlermeldung weist Sie auf das Auftreten einer derartigen Konstellation hin. Verwenden Sie zur Behebung einfach eine andere Operandenadresse.

?GERAET NICHT ANGESCHLOSSEN

Ist ein angesprochenes Peripheriegerät nicht angeschlossen oder ausgeschaltet, wird diese Fehlermeldung ausgegeben. Der .ERROR-Befehl kann bei dieser Meldung einen Abbruch nicht verhindern.

?KEIN LABELSPEICHER MEHR

Diese Meldung weist Sie darauf hin, daß nicht mehr genügend Speicher für Labels vorhanden ist. Die Meldung kann bei der Definition neuer Labels auftreten. Der .ERROR-Befehl kann einen Abbruch des Assemblierens nicht verhindern.

?KEIN MAKRO AUFGERUFEN

Diese Fehlermeldung wird ausgegeben, wenn der Assembler auf einen .ENDM-Befehl trifft, obwohl kein Makro definiert oder aufgerufen wurde. Da die Möglichkeit besteht, daß als Ursache für den Fehler die komplette Speicherverwaltung durcheinander gekommen ist, bricht der Assembler auch bei einem .ERROR-Befehl die Arbeit ab.

?KEIN PROGRAMMSPEICHER MEHR

Dieser Fehler tritt auf, sobald nicht mehr genügend Programmspeicher vorhanden ist. Dies kann bei der Eingabe einer Zeile, beim Laden eines Programms, beim INSERT-, REASS- oder BYTE-Befehl geschehen. Von Makrodefinitionen, die von Diskette eingelesen werden, wird eine Kopie im Programmspeicher erstellt. Es kann also z.B. inmitten einer Makro-Definition diese Meldung erscheinen. Der .ERROR-Befehl kann einen Abbruch des Assemblierens nicht verhindern.

?LABEL BEREITS DEFINIERT

Bei diesem Fehler wurde versucht, ein Label ein zweites Mal zu definieren. Verwenden Sie in diesem Fall einen anderen Labelnamen oder für eine erneute Wertzuweisung den ←-Befehl des Assemblers.

?LABEL NICHT DEFINIERT

In einem mathematischen Ausdruck wurde ein Label aufgerufen, das noch nicht definiert ist. Überprüfen Sie in diesem Fall den Labelnamen, den Sie eingegeben haben, nochmals auf seine Richtigkeit. Beim Assemblieren dürfen nur im 1. Paß nicht definierte Labels verwendet werden. Bei den Pseudo-Befehlen =, ← ,*= und .IF müssen sogar alle verwendeten Labels bereits im 1. Paß definiert sein.

?MAKRO IN MAKRO DEFINIERT

Dieser Fehler entsteht beim Versuch, ein Makro innerhalb eines anderen Makros zu definieren. Verlegen Sie die Definition aus dem Makro heraus. Ursache des Fehlers könnte auch ein fehlender .ENDM-Befehl sein. Eine nachfolgende Makro-Definition fällt dann in die vorangehende. Bei dieser Meldung kann der .ERROR-Befehl den Abbruch des Assemblierens nicht verhindern.

?MAKRO NICHT DEFINIERT

Beim Aufruf eines Makros durch den /-Befehl wurde ein Name verwendet, zu dem kein Makro existiert. Überprüfen Sie den eingegebenen Befehl, und korrigieren Sie den Namen.

?MNEMONIK NICHT VORHANDEN

Diese Meldung wird im Editor ausgegeben, sobald in der Zeile kein oder ein nicht existierendes Mnemonik steht. Das gleiche gilt auch für Pseudobefehle. Der Fehler tritt aber auch auf, wenn das Mnemonik nicht von einem vorangehenden Label oder einer nachstehenden Adresse getrennt ist.

?NICHT ZUGELASSENES GERAET

Bei dieser Fehlermeldung wurde eine Geräteadresse angegeben, die nicht existiert oder die der Befehl nicht zuläßt. Bei Verwendung eines Dateinamen muß die Adresse immer im Bereich 8-11 liegen und ohne Dateinamen im Bereich 3-7.

PASS DIFFERENZ!

Dies ist eine Schlußmeldung des Assemblers, die auf einen Fehler im Programm hinweist. Es wurde im ersten Paß und zweiten Paß ein Maschinenpro-

gramm unterschiedlicher Länge erzeugt. Fehlerhafte Sprünge und Labelwerte sind die Folge. Die Schlußmeldung tritt meist gleichzeitig mit anderen Fehlermeldungen auf. Diese Meldungen werden meist erst im zweiten Paß erzeugt. Im ersten Paß konnte der Objektcode der Zeile noch abgeschätzt werden.

Im zweiten Paß war jedoch eine Berechnung nicht mehr möglich, und der Objektcode fehlt, womit das Programm kürzer wird. Eine zweite mögliche Ursache kann durch folgenden Sachverhalt entstehen: Ist im ersten Paß in einem Operanden ein Label noch nicht definiert, kann der Assembler nicht entscheiden, ob eine 8-Bit- (Zero-Page) oder 16-Bit-Adresse verwendet werden soll (z.B. LDA TEST,X; LDA TABELLE). Er nimmt ersatzweise in diesem Fall eine 16-Bit-Adresse für die Berechnung heran.

Im zweiten Paß muß natürlich an dieser Stelle dann wirklich eine 16-Bit-Adresse verwendet werden. Dies ist normalerweise auch der Fall, da alle später definierten Labels in der Regel Programm-Marken sind.

Wird jedoch z.B. am Ende des Programms erst ein Zero-Page-Label definiert, kann es zu diesem Fehler kommen. Sie sollten sich deshalb angewöhnen, alle Labeldefinitionen zu Beginn des Programms durchzuführen. Fehler werden vermieden, und das Programm wird übersichtlicher.

```
?PROGRAMM BEREITS IM SPEICHER
```

Es wurde versucht, einen neuen Programmnamen in die Programmliste einzutragen, obwohl der Name bereits existiert. Dies kann beim BEGIN-, RENAME, LOAD-, IMPORT-, BYTE- und REASS-Befehl geschehen.

```
?PROGRAMM NICHT IM SPEICHER
```

Diese Meldung teilt mit, daß ein angegebener Programmname im Speicher nicht zu finden ist. Geben Sie am besten mit dem DISPLAY-Befehl alle Programmnamen aus, und stellen Sie fest, welche Programme überhaupt im Speicher abgelegt sind. Ist das gesuchte Programm noch nicht vorhanden, laden Sie es mit LOAD oder fangen es neu mit BEGIN an.

```
?PROGRAMMNAME FEHLT
```

Beim Erscheinen dieser Meldung wurde der Programmname in einem Befehl vergessen. Es muß beachtet werden, daß nur beim SAVE- und ASSEMBLER-Befehl im Editor die Namen entfallen können. Bei allen anderen Befehlen muß ein Programmname angegeben werden.

```
?UNBEKANNTE ADRESSIERART
```

Bei dieser Fehlermeldung wurde ein Adreßteil angegeben, dessen Format nicht vorhanden ist. Ursache hierfür können fehlende Klammern oder Kommata sein. Um solche Fehler zu vermeiden, finden Sie im Anhang B eine Liste aller Adreßformate.

```
?UNERLAUBTE BEFEHLSKOMBINATION
```

Diese Fehlermeldung weist Sie darauf hin, daß Mnemonik und verwendete Adresse in dieser Kombination nicht im Befehlssatz des Prozessors enthalten sind. Zur Beseitigung des Fehlers sollte das Programm noch einmal betrachtet und eine Problemlösung mit erlaubten Befehlen erstellt werden.

```
?UNERLAUBTES ARGUMENT
```

Diese Meldung wird ausgegeben, wenn ein zu hoher oder falscher Wert in einem mathematischen Ausdruck errechnet wurde. Dies können z.B. falsche Bereichsgrenzen beim REASS-Befehl sein. Außerdem erscheint die Meldung, sobald das Ergebnis einer Multiplikation größer als 65535 wird.

```
SUCHE FORSETZEN?
```

Der FIND-Befehl fragt nach jeder gefundenen Programmstelle mit dieser Meldung, ob die Suche fortgesetzt werden soll. Die Tasten <J> und <RETURN> beantworten die Frage mit "Ja", alle anderen verneinen sie und führen zu einem Abbruch des Suchvorgangs.

```
?ZU GROSSE ZAHL
```

Dieser Fehler tritt auf, wenn eine Zahl in einem mathematischen Ausdruck zu viele Ziffern aufweist. Alle Zahlen dürfen nur so groß sein, daß sie noch mit zwei Bytes dargestellt werden können. Für Dezimalzahlen ist der größte Wert 65535.

```
?ZU GROSSER RELATIVER SPRUNG
```

Dieser Fehler tritt nur beim Assemblieren auf. Er weist darauf hin, daß ein zu großer relativer Sprung bei einem Branch-Befehl auftrat. Ein relativer Sprung darf nicht weiter als 128 Byte vorwärts bzw. rückwärts erfolgen. Um

den Fehler zu beheben, sollte der Branch-Befehl durch einen JMP-Befehl ersetzt werden. Wurde im Programm der .ERROR-Befehl gegeben, wird der Objektcode einer fehlerhaften Zeile nicht erzeugt. Die im ersten Paß errechneten Adressen der einzelnen Zeilen stimmen deshalb nicht mit denen des zweiten Paß überein. Bei Verwendung von Labels, die ja im ersten Paß definiert werden, kann es deshalb zur Meldung "ZU GROSSER RELATIVER SPRUNG" kommen, obwohl im fehlerfreien Zustand der Sprung ordnungsgemäß errechnet werden würde.

```
?ZU LANGER NAME
```

Wird ein Programm- oder Dateiname angegeben, der keine oder mehr als 16 Zeichen enthält, erscheint diese Fehlermeldung.

```
?ZU VIELE OFFENE DATEIEN
```

Wird beim Assemblieren die Ausgabe auf ein Peripheriegerät geleitet, öffnet EDASS eine Datei. Das Betriebssystem kann insgesamt nur zehn Dateien verwalten. Da auch BASIC mit dem Befehl OPEN diese Dateiplätze beansprucht, kann es bei mehr als zehn offenen Dateien zu dieser Fehlermeldung kommen. In diesem Fall kann auch der .ERROR-Befehl einen Abbruch nicht verhindern.

```
?ZU VIELE PROGRAMME
```

Der Editor kann bis zu zehn Programme gleichzeitig im Speicher verwalten. Der Versuch, ein weiteres Programm zu beginnen, führt zu dieser Fehlermeldung. Die Befehle BEGIN, LOAD, IMPORT, BYTE und REASS fangen ein neues Programm im Speicher an. Der ERASE-Befehl löscht es wieder.

```
?ZU VIELE VERKETTUNGEN
```

Die Anzahl der Verknüpfungen mit .FILE und .MACRO sind auf 255 begrenzt. Der Versuch einer weiteren Verkettung führt zu dieser Meldung. Der .ERROR-Befehl kann einen Abbruch des Assemblierens nicht verhindern.

```
?ZURUECKSETZEN NICHT ERLAUBT
```

Diese Fehlermeldung tritt auf, wenn Sie versucht haben, den Programmzeiger mit dem *=-Befehl auf eine niedrigere als die aktuelle Position zu setzen, dort

Objektcode erzeugen und diesen gleichzeitig in eine Datei fließen lassen. Auf der Diskette kann nur eine sequentielle Bytefolge gespeichert werden. Ein Zurücksetzen des Programmzeigers stört dieses sequentielle Prinzip noch nicht.

Erst wenn an die neue Position des Programmzeigers Objektcode geschrieben wird, kommt es zu diesem Fehler und zum Abbruch der Assemblierung, der auch nicht vom .ERROR-Befehl verhindert werden kann.

Um einen derartigen Fehler zu beheben, bieten sich mehrere Wege an. Sie können den Objektcode in den Computerspeicher schreiben und von dort mit dem BASIC-Befehl BSAVE speichern. Statt dessen können Sie den Objektcode auch in zwei Dateien schreiben und diese nach Abschluß der Assemblierung mit einem Monitor zusammenflicken. Den wohl elegantesten Weg bietet der .OFF-Befehl. Mit diesem kann ein Offset zwischen dem Programmzeiger und der eigentlichen Schreibadresse erzeugt werden. Ein Umsetzen des Programmzeigers kann damit wieder ausgeglichen werden.

Diskettenfehler

Die einzelnen Diskettenfehler schlagen Sie am besten im Handbuch des Laufwerks nach. Dort erfahren Sie die Ursache der Meldung. Zwei Diskettenfehler, die Ihnen vielleicht häufiger bei der Arbeit begegnen, sollen hier erwähnt werden.

`FILE NOT FOUND`

Beim Auftreten dieses Fehlers haben Sie sich wahrscheinlich beim LOAD-Befehl oder einem anderen Lade-Befehl im Programmnamen vertippt.

`FILE EXISTS`

Hier wurde vergessen, dem Befehl ein Überschreiben des Programms mitzuteilen. Sie erreichen dies erst durch Voranstellen eines Klammeraffen (@) bzw. Paragraphenzeichens (§) vor den Dateinamen.

Kapitel 19

Die mitgelieferten Programme

Auf der EDASS-Diskette befinden sich vier Beispielprogramme, die Sie bei der Erstellung eigener Asssemblerprogramme und beim Umgang mit dem C128 unterstützen. Sie dürfen die Quellprogramme in eigenen Programmen verwenden und beliebig verändern.

Das Software-Interface

Im normalen Betrieb des Computers sind die Geräteadressen 4, 5, 6 und 7 für Drucker reserviert. Als Schnittstelle wird der IEC-Bus verwendet, der auch bei allen Commodore-Druckern eingebaut ist. Es sind jedoch Drucker weit verbreitet, die eine Centronics-Schnittstelle verwenden. Die Verbindung zu derartigen Druckern wird oft über den User-Port und ein Soft-Interface hergestellt.

Das mitgelieferte Programm "INTERFACE" übernimmt diese Aufgabe. Der Objektcode wird mit BLOAD "INTERFACE.O" geladen und mit SYS 4864 gestartet. Danach werden bei Geräteadresse 7 die Daten nicht mehr über den IEC-Bus, sondern über einen am User-Port angeschlossenen Drucker gesendet. In der Anwendung sieht das wie folgt aus:

```
BASIC:   OPEN 1,7          ;KANAL ZUM DRUCKER OEFFNEN
         PRINT#1,...       ;AUSGABE AUF DEN DRUCKER
         CLOSE 1           ;KANAL SCHLIESSEN
EDASS:   .LIST 7           ;LISTING AUF DEN DRUCKER AUSGEBEN
         ...
         .LIST O           ;DIE AUSGABE WIEDER ABSCHALTEN
```

Das Interface-Programme setzt folgende Pin-Belegung am User-Port voraus:

User-Port:	C - L	M	B
Centronics:	Daten	Strobe	Ack.

Die Daten werden parallel über die Pins C bis L des User-Ports übertragen (Daten-Leitungen). Das Anstehen neuer Daten wird dem Drucker über Pin M mitgeteilt (Strobe-Leitung). Der Drucker seinerseits meldet den Empfang der Daten über Pin B (Acknlg-Leitung).

Im Interface-Programm wird mit dem Label CDEV die Geräteadresse der Centronics-Schnittstelle bestimmt. Das eigentliche Programm besteht aus fünf Teilen. Im ersten werden die Betriebssystemvektoren ICKOUT und IBSOUT auf das Soft-Interface gesetzt. Die neue Routine CKOUT, die das Öffnen eines Ausgabekanals übernimmt, befindet sich im nächsten Teil. Die allgemeine Ausgaberoutine BSOUT wird schließlich im dritten Teil neu aufgebaut. Die Initialisierung der Schnittstelle und die Ausgabe eines Bytes erledigen der vierte und fünfte Teil.

Das Interface-Programm verändert zwar die Inhalte der Betriebssystemvektoren, ruft jedoch die Originalroutinen wieder über die alten Vektorwerte auf. Weitere Hilfsprogramme, die ebenfalls die Vektoren verändern, können also gleichzeitig betrieben werden.

Deutsche Umlaute

Der C128 besitzt auf seiner Tastatur standardmäßig Umlaute. Innerhalb des ASCII-Codes liegen diese jedoch nicht an der üblichen Position. Der Umlaut "ü" z.B. hat beim C128 den Code 189 und im ASCII-Code die Nummer 125.

Bei ASCII-Druckern ergeben diese Abweichungen bei der Ausgabe von Listings oder Texten unangenehme Effekte. Das Programm "UMLAUTE" wandelt bei der Ausgabe auf einen Drucker (Geräteadresse 4-7) die Codes der Umlaute vom Commodore-Code in den richtigen ASCII-Code um. Der Objektcode des Programms wird mit BLOAD "UMLAUTE.O" geladen und mit SYS 5120 gestartet.

Das Programm verändert den Inhalt des Vektors IBSOUT. Der alte Inhalt wird zwischengespeichert. Nach Abschluß der Umwandlung wird auch die alte Ausgaberoutine angesprochen. So ist z.B. ein paralleler Betrieb des Soft-Interfaces und dieses Umwandelprogramms möglich. Beachtet werden muß nur, daß zuerst das Interface- und dann das Umwandelprogramm gestartet wird.

Makros zur 16-Bit-Arithmetik

Das Programm "16-BIT-MAKROS" enthält eine Sammlung von Makros zur 16-Bit-Arithmetik. Diese übernehmen Aufgaben wie die Addition oder Division zweier 16-Bit-Zahlen. Das Quellprogramm wird beim Assemblieren mit dem Hauptprogramm verknüpft. Die einzelnen Makros werden damit definiert. Die Argumente der mathematischen Operationen, die in den Makros durchgeführt werden, müssen im Speicher in der Reihenfolge LO-/HI-Byte

abgelegt sein. Als Parameter werden dem Makro die Adressen der Speicherzellen mitgeteilt. Noch ein Wort zu den Parametern: Der erste Parameter ist grundsätzlich die Adresse des Ergebnisses, die nachfolgenden Parameter die Adressen der Argumente. Alle Makros verändern nur den Akkumulatorinhalt.

Es folgt jetzt eine kurze Beschreibung der einzelnen Makros. Es werden die Operationen und die Eingangsparameter beschrieben. Als Überschrift für jedes Makro werden der Name, unter dem dieses aufgerufen wird, und die Parameter des Makros genannt.

ADD (ERG, SUM1, SUM2) – Addition zweier Zahlen

Dieses Makro addiert zwei Zahlen. Wie bei einer normalen Addition zeigt das Carry-Flag einen Übertrag und das Overflow-Flag einen Überlauf an.

ADDDATA (ERG, SUM, SUMDATA) – Additon eines festen Wertes

Im Gegensatz zur vorherigen Addition wird hier nicht der Inhalt zweier Speicherzellen, sondern der Inhalt einer Speicherzelle und ein fester Wert addiert. Der dritte und letzte Parameter enthält den Zahlenwert. Auch hier zeigen das Carry- und das Overflow-Flag weitere Informationen über den Verlauf der Addition an.

SUB (ERG, MIN, SUB) – Subtraktion zweier Zahlen

Dieses Makro subtrahiert zwei Zahlen. Es wird der dritte Parameter vom zweiten subtrahiert. Das Carry- und des Overflow-Flag zeigen weitere Ergebnisse der Subtraktion an.

SUBDATA (ERG, MIN, SUBDATA) – subtrahiere einen festen Wert

Bei dieser Subtraktion wird vom Inhalt eines Speicherzellenpaars (eine 16-Bit-Zahl) ein fester Wert subtrahiert. Dieser wird als dritter Parameter übergeben.

LOGAND (ERG, ARG1, ARG2) – logisch UND-verknüpfen

Durch dieses Makro werden zwei Zahlen logisch Und-verknüpft.

LOGOR (ERG, ARG1, ARG2) – logisch ODER-verknüpfen

Diese Makro verknüpft zwei Zahlen logisch "oder" miteinander.

LOGEXOR (ERG, ARG1, ARG2) – logisch EXOR-verknüpfen

Wie die zwei vorangegangenen Makros führt auch dieses eine logische Verknüpfung durch. Die zwei Zahlen werden miteinander "exor" verknüpft.

MULTI (ERG, FAK1, FAK2) – Multiplikation zweier Zahlen

Dieses Makro multipliziert zwei 16-Bit-Zahlen miteinander. Das Ergebnis umfaßt ebenfalls 16 Bit. Da bei diesen Zahlbegrenzungen Ergebnisse mit mehr als 16 Stellen auftreten können, wird mit dem Carry-Flag ein Fehler angezeigt. Ist am Ende der Routine das C-Flag gesetzt, ist das Ergebnis größer als 16 Bit. Bei der Routine ist zu beachten, daß die Inhalte der Speicherzellen des Faktors FAK1 verändert werden.

DIV (ERG, DVDND, DVSR) – Division zweier Zahlen

Mit diesem Makro werden zwei Zahlen dividiert. Ist der Divisor null, kann eine Division nicht durchgeführt werden. Dieser Zustand wird am Ende der Routine durch ein gesetztes Carry-Flag angezeigt. Bei der Division werden die Speicherzellen des Dividenden (DVDND) verändert. Sie enthalten am Ende den Rest der Division.

SQR (ERG, QUAD) – Quadratwurzel einer Zahl

Dieses Makro zieht aus einer Zahl die Quadratwurzel. Das Ergebnis enthält den ganzzahligen Anteil. Hier ist auch wieder zu beachten, daß die Quadratzahl (QUAD) verändert wird. Sie zeigt den negativen Rest der Quadratzahl an.

INVERT (ARG) – Vorzeichenwechsel

Dieses Makro ändert, gemäß der Zweier-Komplement-Darstellung, das Vorzeichen einer Zahl. Aus einer positiven wird eine negative und aus einer negativen eine positive Zahl.

CMPRE (ARG1, ARG2) – vergleicht zwei Zahlen miteinander

Den Vergleich zweier 16-Bit-Zahlen führt dieses Makro durch. Am Ende kann wie gewohnt abgefragt werden, ob die zwei Zahlen identisch sind, welche größer bzw. kleiner war.

CMPDATA (ARG, CDATA) – Vergleich einer Zahl mit einem Wert

Dieses Makro vergleicht eine Zahl mit einem festen Wert. Auch hier kann wie gewohnt entschieden werden, ob die Zahl mit dem Wert identisch, kleiner oder größer ist.

SHFTL (ARG) – Linksverschieben einer Zahl

Das Verschieben einer 16-Bit-Zahl erfolgt mit diesem Makro. Eine wichtige Bedeutung kommt diesem Makro zu angesichts der Tatsache, daß das Linksverschieben einer Multiplikation mit zwei entspricht.

SHFTR (ARG) – Rechtsverschieben einer Zahl

Dieses Makro schiebt die Bits einer Zahl nach rechts. Dies entspricht einer Division durch zwei.

PUSH (ARG) – eine Zahl auf den Stack legen

Der Inhalt einer Zahl wird auf dem Stack zwischengespeichert. Es wird zuerst das HI- und dann das LO-Byte abgelegt. Dies entspricht der Reihenfolge, in der auch der JSR-Befehl eine Rücksprungadresse auf den Stack legt. Diese Tatsache kann in Anwendungen ausgenutzt werden.

PULL (ARG) – eine Zahl vom Stack lesen

Eine mit dem PUSH-Makro gespeicherte Zahl wird mit diesem Makro wieder gelesen. Zuerst wird das LO- und dann das HI-Byte vom Stack gelesen.

PUSHALL – Retten der Registerinhalte auf den Stack

Dieses Makro rettet das Statusregister, den Akkumulator, das X- und das Y-Register auf den Stack. Die Reihenfolge entspricht der hier angegebenen Aufzählung. Durch die Routine wird der Inhalt des Akkumulators verändert.

PULLALL – Registerinhalte wieder vom Stack lesen

Mit diesem Makro werden Registerinhalte wieder vom Stack gelesen. Zuerst wird das Y-, dann das X-Register, danach der Akkumulator und schließlich das Statusregister gelesen.

COPY (TO, FROM) – Kopieren einer Zahl

Dieses Makro überträgt den Inhalt der Zahl FROM in die Zahl TO.

COPYDATA (TO, DATA) – einer Zahl einen Wert zuweisen

Im Gegensatz zur vorherigen Routine wird mit diesem Makro ein fester Zahlenwert kopiert. Das Makro wird grundsätzlich verwendet, um einer 16-Bit-Zahl erstmals einen Wert zuzuweisen.

EXCHG (ARG1, ARG2) – Austausch zweier Zahlen

Dieses Makro vertauscht die Inhalte der angegebenen Speicherzellen. Zur Demonstration der Makros sehen Sie unten ein komplettes Beispielprogramm, das die Rechnung "SQR (600/100+30)" durchführt. Das Ergebnis wird durch das Makro CMPDATA geprüft, und auf dem Bildschirm wird die Korrektheit angezeigt.

```
        .BANK       15
        *=                              $1300
        .OBJ        M
BSOUT   =           $FFD2               ;BSOUT AUSGABEROUTINE
ZAHL1   =           250                 ;ERSTE RECHENZAHL
ZAHL2   =           252                 ;ZWEITE RECHENZAHL
ZAHL3   =           254                 ;DRITTE RECHENZAHL
        .FILE       "16-BIT-MAKROS"     ;ALLE MAKROS DEFINI.
        /COPYDATA   (ZAHL1,600)         ;ZAHL1=600
        /COPYDATA   (ZAHL2,100)         ;ZAHL2=100
        /DIV        (ZAHL3,ZAHL1,ZAHL2) ;ZAHL3=ZAHL1/ZAHL2
        /ADDDATA    (ZAHL2,ZAHL3,30)    ;ZAHL2=ZAHL3+30
        /SQR        (ZAHL1,ZAHL2)       ;ZAHL1=SQR(ZAHL2)
        /CMPDATA    (ZAHL1,6)           ;VERGLEICH ZAHL1 MIT 6
        BNE         ENDE                ;GLEICH ?, JA =>
        LDA         #"="                ;AKKU. = ASCII VON "="
        JSR         BSOUT               ;DAS ZEICHEN AUSGEBEN
ENDE    RTS                             ;PROGRAMM VERLASSEN
        .END
```

Makros zur strukturierten Programmierung

Bei der Programmierung in Assemblersprache muß jede kleine Schleife oder Abfrage aus einzelnen Teilen zusammengesetzt werden. Die Übersichtlichkeit geht schnell verloren. Im Programm "STRUK-MAKROS" finden Sie eine Sammlung von Makros, die Strukturen wie Schleifen und Verzweigungen bereits fertig enthalten. Zum Aufbau einer FOR-Schleife wird einem Makro als Parameter nur die Speicherzelle des Zählers, der Anfangs- und Endwert genannt.

Das Ende der Schleife wird durch das Makro NEXT begrenzt. Den korrekten Aufbau der Schleife durch Maschinenbefehle übernimmt jetzt das Makro. Die angebotenen Strukturen dürfen beliebig ineinander geschachtelt werden. Einzige Ausnahme bilden hier die CASE-Strukturen. Sie dürfen in sich selbst nicht geschachtelt werden.

Bei der WHILE- und REPEAT-Schleife sowie bei der IF-Verzweigung wird als Parameter ein Wert verlangt, der eine Bedingung auswählt. Diese Bedingungen entsprechen den Verzweigungsbedingungen der Branch-Befehle. Es ergeben sich folgende Zahlenwerte:

```
EQ    1          NE          -1
CS    2          CC          -2
VS    3          VC          -3
MI    4          PL          -4
```

Die jeweils inverse Bedingung trägt auch das inverse Vorzeichen. Um die Parametervorgabe zu vereinfachen, werden im Programm "STRUK-MAKROS" die Labels EQ, CS, VS usw. definiert. Ihnen werden die entsprechenden Werte zugewiesen.

```
/IF   (3)          entspricht      /IF          (VS)
```

Für die interne Verwaltung der Labels werden des weiteren die Labels F, R, W, I und C definiert. Zur Zwischenspeicherung von Werten wird das Label X benutzt. Es folgt eine Beschreibung der verschiedenen Strukturen im einzelnen. Als Überschriften sind der Name des Makros und die geforderten Parameter angegeben.

FOR (CNT, FROM, TO) – Start einer FOR-Schleife

Mit CNT wird die Adresse einer 16-Bit-Zahl genannt. Diese wird im Speicher in der Reihenfolge LO-, HI-Byte verwaltet. Den Startwert enthält der Para-

meter FROM und den Endwert der Parameter TO. Um die Programmstruktur kurz zu halten, beträgt der maximale Endwert 65534 ($FFFE).

NEXT – Ende einer FOR-Schleife

Dieses Makro markiert das Ende einer FOR-Schleife. Hier wird der Zähler um eins erhöht und die Endbedingung überprüft. Der Schleifenanfang und das Schleifenende dürfen im Programm beliebig weit auseinander liegen.

FORX (FORM, TO) – Start einer FOR-Schleife mit dem X-Register

Dieses Makro baut eine spezielle FOR-Schleife mit dem X-Register auf. Dementsprechend ist die Schleife auch sehr schnell.

NEXTX – Ende einer FOR-Schleife mit dem X-Register

Dieses Makro beendet eine mit FORX geöffnete Schleife. Um die Struktur kurz zu halten, wird der Sprung zum Schleifenanfang durch einen Branch-Befehl übernommen. Schleifenanfang und Schleifenende dürfen deshalb nur ca. 120 Bytes auseinander liegen.

FORY (FORM, TO) – Start einer FOR-Schleife mit dem Y-Register

Ähnlich wie bei der vorherigen Schleife wird auch bei dieser ein Register, hier das Y-Register, als Schleifenzähler benutzt.

NEXTY – Ende einer FOR-Schleife mit dem Y-Register

Dieses Makro beendet eine mit dem Makro FORY geöffnete Schleife. Der Rücksprung zum Schleifenanfang wird über einen Branch-Befehl hergestellt, wodurch der Schleifenanfang maximal ca. 120 Bytes entfernt sein darf.

REPEAT – Anfang einer REPEAT-Schleife

Dieses Makro legt den Start einer REPEAT-Schleife fest.

UNTIL (COND) – Ende einer REPEAT-Schleife

Dieses Makro beendet eine Repeat-Schleife. Als Parameter wird eine Bedingung übergeben. Entsprechend dieser Bedingung wird ein Branch-Befehl in

das Programm eingebaut. Vor dem UNTIL-Makro muß also z.B. ein Vergleich oder eine andere Operation stehen, die das entsprechende Flag beeinflußt. Der Schleifenanfang (REPEAT) und das Schleifenende (END) dürfen beliebig weit auseinanderliegen.

WHILE (COND) – Anfang einer WHILE-Schleife

Mit diesem Makro wird eine WHILE-Schleife begonnen. Dem Makro müssen mehrere Befehle folgen, die die Endbedingung errechnen, also z.B. ein Vergleich. Am Ende dieser Befehlsgruppe wird durch einen Branch-Befehl, dessen Bedingung als Parameter übergeben wird, geprüft, ob die Schleifenbedingung noch erfüllt ist.

DO – Schleifenrumpf einer WHILE-Schleife

Durch dieses Makro wird zum einen die Befehlsgruppe zum Überprüfen der Endbedingung begrenzt, zum andern wird damit der Schleifenrumpf der WHILE-Schleife eingeleitet.

WEND – Ende einer WHILE-Schleife

Dieses Makro markiert das Ende einer WHILE-Schleife.

IF (COND) – IF-Teil einer Verzweigung

Dieses Makro führt eine Verzweigung durch. Die als Parameter übergebene Bedingung wird in einen Branch-Befehl umgesetzt. Dem Makro muß also eine Operation vorausgehen, die das entsprechende Flag beeinflußt. Die dem IF-Makro folgenden Befehle stellen den THEN-Zweig der Verzweigung dar.

ELSE – ELSE-Zweig einer Verzweigung

Dieses Makro leitet den ELSE-Zweig der Verzweigung ein. Alle nachfolgenden Befehle stellen diesen dar. Ein ELSE-Zweig ist nicht unbedingt erforderlich. Es ist auch eine reine IF-THEN-Verzweigung erlaubt.

ENDIF – Ende einer Verzweigung

Durch dieses Makro wird das Ende einer Verzweigung markiert. Das Makro muß immer eingegeben werden.

CASE (FLAG) – Anfang einer CASE-Struktur

Mit diesem Makro wird eine CASE-Struktur aufgebaut. Als Parameter wird die Adresse einer Speicherzelle übergeben. Deren Inhalt wird zu Beginn in den Akkumulator geladen.

SWITCH (VAL) – Einzelner Zweig einer CASE-Struktur

Als Parameter wird bei diesem Makro der Vergleichswert für den Zweig übergeben. Trifft dieser Wert zu, werden die nachfolgenden Befehle als Zweig ausgeführt. Im anderen Fall werden diese durch einen Branch-Befehl übersprungen.

BREAK – Ende eines Zweigs bei einer CASE-Struktur

Dieses Makro markiert das Ende eines Zweigs. Das Makro muß angegeben werden, da sonst die nachfolgende Bedingung geprüft würde.

DEFAULT – Default-Zweig einer CASE-Struktur

Mit diesem Makro wird der Default-Zweig einer CASE-Struktur eingeleitet. Das Makro muß immer eingegeben werden, um einen korrekten Aufbau der Struktur zu ermöglichen. Wird kein Default-Zweig benötigt, können nachfolgende Befehle entfallen. Der Defaultzweig fällt dann mit dem Ende der CASEStruktur zusammen. Am Ende des Default-Zweigs ist kein BREAK-Makro erforderlich.

ENDCSE – Ende einer CASE-Struktur

Dieses Makro markiert das Ende einer CASE-Struktur. Es darf nicht entfallen.

CASEX (FLAG) – Anfang einer CASE-Struktur über das X-Register

Gegenüber der vorherigen CASE-Struktur wird hier der Inhalt der angegebenen Speicherzelle in das X-Register geladen.

SWITCHX (VAL) – Zweig einer CASE-Struktur über das X-Register

Dieses Makro startet den Zweig einer CASE-Struktur, die über das X-Register aufgebaut wird. Die Makros BREAK, DEFAULT und ENDCSE dürfen auch hier direkt verwendet werden.

CASEY (FLAG) – Anfang einer CASE-Struktur über das Y-Register

Dieses Makro baut eine CASE-Struktur über das Y-Register auf.

SWITCHY (VAL) – Zweig einer CASE-Struktur über das Y-Register

Diese Makro leitet den Zweig einer CASE-Struktur ein, die durch das Y-Register aufgebaut wird.

Die jeweils zusammengehörigen Makros und deren Bedeutung sind nochmals in folgender Tabelle zusammengestellt:

FOR (Schleifenrumpf) **NEXT**	**FORX** (Schleifenrumpf) **NEXTX**	**FORY** (Schleifenrumpf) **NEXTX**
REPEAT (Schleifenrumpf) **UNTIL**		
WHILE (Routine zum Errechnen der Bedingung) **DO** (Schleifenrumpf) **WEND**		

IF
(THEN-Zweig)
ELSE
(ELSE-Zweig)
ENDIF
	IF
	(THEN-Zweig)
	ENDIF	

CASE **SWITCH** (einzelner Zweig) **BREAK** . . **DEFAULT** (Default-Zweig) **ENDCSE**	**CASEX** **SWITCHX** (einzelner Zweig) **BREAK** . . **DEFAULT** (Default-Zweig) **ENDCSE**	**CASEY** **SWITCHY** (einzelner Zweig) **BREAK** . . **DEFAULT** (Default-Zweig) **ENDCSE**

Es folgt jetzt ein Beispiel, das das gesamte Alphabet ausgibt. Das Problem wird mit einer FOR-, einer REPEAT- und einer WHILE-Schleife gelöst.

```
          .BANK     15
          *=                              $1300
          .OBJ      M
BSOUT     =         $FFD2                 ;BSOUT AUSGABEROUTINE
ZAEHLER   =         250                   ;SCHLEIFENZAEHLER
          .FILE     "STRUK-MAKROS"        ;ALLE MAKROS DEFINI.
          /FORX     (ZAEHLER,"A","Z")     ;SCHLEIFENSTART
          TXA
          JSR       BSOUT
          /NEXTX                          ;SCHLEIFENENDE
          RTS                             ;PROGRAMM VERLASSEN
          .END


          .BANK     15
          *=        $1300
          .OBJ      M
BSOUT     =         $FFD2                 ;BSOUT AUSGABEROUTINE
          .FILE     "STRUK-MAKROS"        ;ALLE MAKROS DEFININIEREN
          LDA       #"A"
          /REPEAT                         ;SCHLEIFENSTART
          JSR       BSOUT
          CLC
          ADC       #1
          CMP       #"Z"+1
          /UNTIL    (EQ)                  ;ENDBEDINGUNG
          RTS
          .END


          .BANK     15
          *=        $1300
          .OBJ      M
BSOUT     =         $FFD2                 ;BSOUT AUSGABEROUTINE
          .FILE     "STRUK-MAKROS"        ;ALLE MAKROS DEFININIEREN
          LDA       #"A"
          /WHILE    (NE)                  ;SCHLEIFENSTART
          CMP       #"Z"+1
          /DO                             ;START DES RUMPFES
          JSR       BSOUT
          CLC
          ADC       #1
          /WEND                           ;SCHLEIFENENDE
          RTS                             ;PROGRAMM VERLASSEN
          .END
```

Bei der Verwendung der Makros zur strukturierten Programmierung müssen Sie selbst auf einen korrekten Einsatz und eine ordnungsgemäße Schachtelung achten. Der Assembler und die Makros können keine Überprüfung vornehmen. Einen Fehler merken Sie erst an einer fehlerhaften Programmausführung.

Kapitel 20

Programmierhinweise

Tips für die Programmerstellung

Hier sollen ein paar Tips gegeben werden, die beim Schreiben von Assemblerprogrammen sehr nützlich sind und die zu einem leicht verständlichen und sauber aufgebauten Programm führen.

1. Labels sollten Namen erhalten, die auf ihre Bedeutung schließen lassen. Für einen Programmanfang wählt man z.B. sinnvollerweise "START" oder z.B. für den Anfang einer Schleife "LOOP".

2. Unterprogramm- und Tabellenadressen sollten immer durch Labels vertreten werden. Dies gilt vor allem für ROM-Adressen. Wenn sie im Programm verwendet werden, muß nicht lange der Wert irgendwo nachgeschlagen werden, sondern es genügt einfach, den Namen des Labels, das die Adresse vertritt, einzugeben. Hier ist es auch wichtig, daß sinnvolle Namen gewählt werden.

3. Assemblerprogramme sollten immer kommentiert werden. Nur so ist es möglich, auch Monate später den Aufbau eines Programms wieder schnell zu erkennen.

4. Mit dem *=-Befehl können im Programm Speicherbereiche innerhalb des Objektcodes reserviert werden. Dies geschieht durch die Befehlsfolge:

   *= *+(Bereichsgröße)

   Diese Möglichkeit ist sehr nützlich, wenn Speicherplätze für programminterne Flags, Variablen oder Puffer benötigt werden.

5. Bei Branch-Befehlen wird im Programm nicht die Größe des relativen Sprungs angegeben, sondern die Adresse, zu der gesprungen wird. Damit nicht für jeden kleinen Sprung im Programm ein neues Label definiert werden muß, kann die Sprungadresse aus dem Programmzeiger errechnet werden. Soll z.B. zu einem 4 Byte entfernten Punkt gesprungen werden, muß als Adresse einfach *+4 eingegeben werden. Das folgende kleine Programmbeispiel verdeutlicht diesen Sachverhalt noch einmal:

```
        INC   POINT                 INC     POINT
        BNE   ENDE                  BNE     *+4
        INC   POINT+1               INC     POINT+1
ENDE    RTS                         RTS
```

Im Beispiel steht der Programmzeiger beim Assemblieren des Branch-Befehls auf dem BNE-Befehl, um ein Byte versetzt auf dem Programmpunkt, der die Größe des relativen Sprungs angibt, noch ein Byte versetzt auf dem INC-Befehl, ein weiteres Byte versetzt auf dem Adreßteil des INC-Befehls und schließlich nach dem vierten Byte auf dem RTS-Befehl. Durch diese Methode müssen nicht so viele Labels definiert werden, womit, wie hier im Beispiel, das Programm übersichtlicher wird. Diese Art der Adreßbestimmung sollte nur bei sehr kurzen Sprüngen Einsatz finden.

Programme für den C64-Modus

Da die Maschinenbefehle des C64 und C128 identisch sind, können mit EDASS auch Programme für den C64-Modus geschrieben werden. EDASS selbst läuft natürlich nicht in diesem Modus. Das Programm wird normal mit dem Editor eingegeben. Beim Assemblieren schreiben Sie den Objektcode in eine Programmdatei. Zum Ausführen des Programms wechseln Sie mit dem BASIC-Befehl GO64 in den C64-Modus über. Dort lesen Sie das Maschinenprogramm wieder mit LOAD"...",8,1 in den Speicher ein. Bei Fehlern muß zum Ändern in den C128-Modus zurückgeschaltet und EDASS neu gestartet werden.

Technische Informationen zu EDASS

Speicherbelegung und Aufteilung

Das Programm EDASS belegt folgende Speicherbereiche:

Bank 0	$03E4 - $03EF	(EDASS-Aufruf)
	$1A00 - $1BFF	(ROM-Sprungtabelle)
	$F000 - $FEFF	(Routinen in Common Area)
Bank 1	$B400 - $FEFF	(Hauptprogramm)

Bei der Bearbeitung eines Befehls wird die Common Area auf 4KByte vergrößert und an beide Speicherenden gelegt. Diese Tatsache muß beachtet werden, wenn Zusatzprogramme, wie z.B. ein Software-Interface, parallel mit EDASS betrieben werden. Damit auch solche Programme mit EDASS betrieben werden können, ist es erforderlich, daß der Speicher im freien Bereich von $1300 bis $1BFF belegt wird. BASIC ruft nämlich ROM-Routi-

nen immer mit eingeschalteter Bank 0 auf. Routinen, die Betriebssystemvektoren verbiegen, müssen deshalb auch in Bank 0 liegen. Damit diese Erweiterungen auch beim Betrieb von EDASS korrekt angesprochen werden, muß das ROM von Bank 0 aus angesprochen werden. Aus diesem Grund wird dieser Speicherbereich in Bank 0 belegt. Diese Speicherverteilung ist auch Ausgangspunkt für eine zweite Version von EDASS die mitgeliefert wird. Diese ist in der Bedienung identisch, belegt jedoch den Bereich von $1A00-$1BFF nicht. Erweiterungen der beschriebenen Art dürfen hier nicht betrieben werden. Geladen wird diese Version mit RUN"EDASS 128.2".

Wenn schon Zusatzprogramme erwähnt wurden, gleich zu den von EDASS versetzen Vektoren bzw. Vektoren, die nicht verändert werden dürfen:

IMAIN	-	$0302	(von EDASS auf neuen Wert gesetzt)
CLTVEC	-	$0334	(darf nicht verändert werden)
SHFVEC	-	$0336	(darf nicht verändert werden)
ESCVEC	-	$0338	(darf nicht verändert werden)

Bei Zusatzprogrammen ist weiter zu beachten, daß bei der Abarbeitung von EDASS-Befehlen die Prekonfiguration-Register der Speicherverwaltung einen anderen Wert als bei BASIC enthalten.

Nun zur Speicheraufteilung von EDASS. In der Zero-Page werden nahezu alle Speicherplätze, in denen der C128 keine dauerhaften Werte speichert, von EDASS benutzt. Der Programmtext wird in Bank 1 abgelegt. Diese Bank muß deshalb zwischen BASIC-Variablen und Assemblerprogrammen aufgeteilt werden. In Bank 0 sind die Labels abgelegt. Diese teilen sich die Bank mit BASIC-Programmen.

Der BASIC-Befehl BANK und die Assembleranweisung .BANK verwenden dieselbe Speicherzelle, um die Banknummer festzulegen. Die Befehle REASS, BYTE und GO nutzen diese Vorgabe. Die Bank kann also mit dem BASIC-Befehl BANK neu gesetzt oder vom Assembler direkt übernommen werden. Bei dessen Aufruf wird ohne .BANK-Anweisung automatisch auf Bank 15 geschaltet.

Datenformate

Zuerst zur Speicherung der Assemblerprogramme: Der Name des Programms wird als erstes im Speicher abgelegt. Er kann max. 16 Zeichen umfassen und wird mit einem Null-Byte abgeschlossen. Der sich daraus ergebende Maximalbedarf von 17 Bytes wird immer reserviert. Direkt danach folgt der Programmtext.

Eine einzelne Zeile ist wie folgt aufgebaut:

Byte 0 – Länge der Zeile (einschließlich Schluß-Null)
Byte 1 – LO-Byte der Zeilennummer
Byte 2 – HI-Byte der Zeilennummer
ab Byte 3 – Zeilentext:

- Labelname in ASCII-Zeichen
- der in ein Token umgewandelte Befehl (der Wert ist zur Kennzeichnung größer gleich 128)
- Adreßteil des Befehls in ASCII-Zeichen
- Byte mit dem Wert 1 als Markierung für den Kommentar (diese Marke wird bei der Ausgabe einen Strichpunkt verwandelt)
- Kommentar in ASCII-Zeichen

Byte n – ein Null-Byte als Endmarke der Zeile

Die Zeilennummern der Zeilen werden vom Editor in Einer-Schritten aufwärtsgezählt. Beim Einfügen neuer Zeilen werden die Nummern sofort korrigiert. Das Längenbyte der Zeile dient gleichzeitig als Endmarke für das gesamte Programm. Eine Null an dieser Stelle signalisiert das Ende des Programmtextes. Bis zu zehn Programme stehen im Speicher hintereinander. EDASS besitzt für den Zugriff eine Tabelle mit den Startadressen. Werden Makrodefinitionen von Diskette eingelesen (über .FILE-Befehl) muß eine Kopie des Makros im Speicher abgelegt werden. Diese wird ebenfalls in den Programmspeicher geschrieben. Von Makrodefinitionen, die in Programmen vorgenommen werden, die bereits im Speicher stehen, werden natürlich keine Kopien erstellt. Bei der Speicherung der Programme auf Diskette wird, um Speicherplatz zu sparen, nur der Zeilentext mit Endmarke in die Datei geschrieben. Zeilenlänge und Zeilennummer entfallen. Jetzt zur Speicherung der Labels. Ein solches ist wie folgt aufgebaut:

Byte	0	–	LO-Byte des Labelwertes
Byte	1	–	HI-Byte des Labelwertes
Byte	2	–	Bit 0 –5 Länge des Namens Bit 6 Flag für ein Makro Bit 7 Flag für ein globales Label
Byte	3	–	Zähler (1 – 255) zur Unterscheidung der Herkunft der Labels. Bei jedem .FILE-Aufruf bzw. Makro-Aufruf wird den Labels eine neue Nummer gegeben. Bei globalen Labels ist dieses Byte bedeutungslos.
ab Byte	4	–	der Name des Labels aus ASCII-Zeichen

Den beiden höchstwertigen Bits von Byte 2 kommen außer den angegebenen Funktionen noch weitere Aufgaben zu. Folgende Kombinationen haben entsprechende Bedeutung:

Bit 7	Bit 6	Bedeutung
0	0	normales Label
0	1	es handelt sich um ein Makro
1	0	ein globales Label
1	1	dieses Label wurde vom Reassembler erzeugt

Byte 3, das zur Unterscheidung der Herkunft von Labels dient, kann nur Werte von 1 bis 255 annehmen. Das Hauptprogramm und der Direktmodus besitzen die Nummer Eins. Mit dem KILL-Befehl wird bei einem Label die globale- und Makro-Marke entfernt und dieses Herkunftsbyte auf den Wert Null gesetzt. Das Label ist damit nicht mehr durch EDASS zu erreichen.

Der PUSH-Befehl speichert die gesamte Labelinformation hintereinander in einer Datei.

Bei einem Makro entspricht der Labelname dem Makronamen, und als Labelwert wird die Startadresse des Makrotextes im Speicher abgelegt. Diese Adresse kann sowohl in ein im Speicher befindliches Programm zeigen als auch in die Kopie einer Makrodefinition.

Zuletzt zu den in der Datei "PARAMETER" (erzeugt mit STORE) gespeicherten Daten. Diese haben folgende Bedeutung:

	Byte	0	–	Zeilenbreite des 40-Zeichen-Editors
	Byte	1	–	Markierung der Eingabezeile (40-Zeichen)
	Byte	2	–	Markierung der Eingabezeile (80-Zeichen)
	Byte	3	–	Einleitungszeichen für die Befehle
von	Byte	4	–	die Funktionstexte in der Form, wie sie auch
bis	Byte	259	–	im Speicher abgelegt sind
	Byte	260	–	LO-Byte des Anfangs des Labelspeichers
	Byte	261	–	entsprechendes HI-Byte
	Byte	262	–	LO-Byte des Anfangs des Programmspeichers
	Byte	263	–	entsprechendes HI-Byte

Ist nach Lesen von Byte 0 bereits das Dateiende erreicht oder ein Fehler aufgetreten, wird die Datei nicht eingelesen und eine Standardbelegung angenommen.

Anhang A

Lösungen zu den Aufgaben

Zu allen Aufgaben gibt es mehrere Lösungen. Es kann nicht jede einzelne Lösung bedacht werden, und bestimmt würden Sie auch dann wieder neue finden. Es wird deshalb nur eine Musterlösung erläutert und abgedruckt. Solange Ihre Programmlösung das von der Aufgabenstellung gewünschte Ergebnis liefert, ist sie richtig. Ob dies dann auch die eleganteste Lösung ist, spielt keine Rolle.

Aufgabe 2.1 In den Bildschirmspeicher muß jetzt der Code eines Sterns geschrieben werden. In Anhang H ist eine Tabelle der Bildschirmcodes abgedruckt. Aus dieser können Sie entnehmen, daß ein Stern den Code 42 hat. Der Akkumulator muß also mit 42 geladen werden. Dieser Wert wird dann mit dem Befehl STA in den Bildschirmspeicher kopiert.

Hierzu das vollständige Programm:

```
*=      $1300                   ;STARTADRESSE IST $1300
.OBJ    M                       ;OBJEKTCODE IN DEN SPEICHER
.BANK 15                        ;SPEICHERAUFBAU WAEHLEN
LDA     #42                     ;AKKU. MIT CODE VON "*" LADEN
STA     1024                    ;DIESEN WERT IN DEN BILDSPEICHER
LDA     #7                      ;AKKU. MIT CODE VON GELB LADEN
STA     55296                   ;DIESEN WERT IN DEN FARBSPEICHER
RTS                             ;DAS PROGRAMM VERLASSEN
.END
```

Nachdem das Programm assembliert wurde, kann es mit !GO $1300 gestartet werden. Auf dem Bildschirm erscheint links oben ein gelber Stern

Aufgabe 2.2 Um das erste Herzchen auf den Bildschirm zu schreiben, muß der Bildschirmcode eines Herzchens in die Speicherzelle 1024 und der Farbwert in die Speicherzelle 55296 geschrieben werden. Das zweite Herzchen muß rechts neben dem ersten erscheinen. Durch den zeilenweisen Aufbau des Bildschirmspeichers sind dies folglich die Adressen 1025 und 55297. Den Bildschirmcode eines Herzchens können Sie wieder in Anhang H nachschlagen. Dort finden Sie auch eine Tabelle der Farbcodes.

Um das Programm kurz zu halten, wird für jedes Herzchen der Akkumulator nicht neu geladen. Der Speicherbefehl STA kopiert nur den Inhalt des Akkumulators, womit dieser unverändert bleibt. In einem zweiten Speicherbefehl kann dieser nochmals in den Bildschirmspeicher kopiert werden.

```
*=      $1300           ;STARTADRESSE IST $1300
.OBJ    M               ;OBJEKTCODE IN DEN SPEICHER
.BANK 15                ;SPEICHERAUFBAU WAEHLEN
LDA     #83             ;AKKU. MIT CODE VON "♥" LADEN
STA     1024            ;DIESEN WERT IN DEN BILDSPEICHER
STA     1025            ;NOCHMALS IN DEN BILDSPEICHER
LDA     #2              ;AKKU. MIT CODE VON ROT LADEN
STA     55296           ;DIESEN WERT IN DEN FARBSPEICHER
STA     55297           ;NOCHMALS IN DEN FARBSPEICHER
RTS                     ;DAS PROGRAMM VERLASSEN
.END
```

Zur Verdeutlichung werden die Datentransfers in einem Diagramm dargestellt:

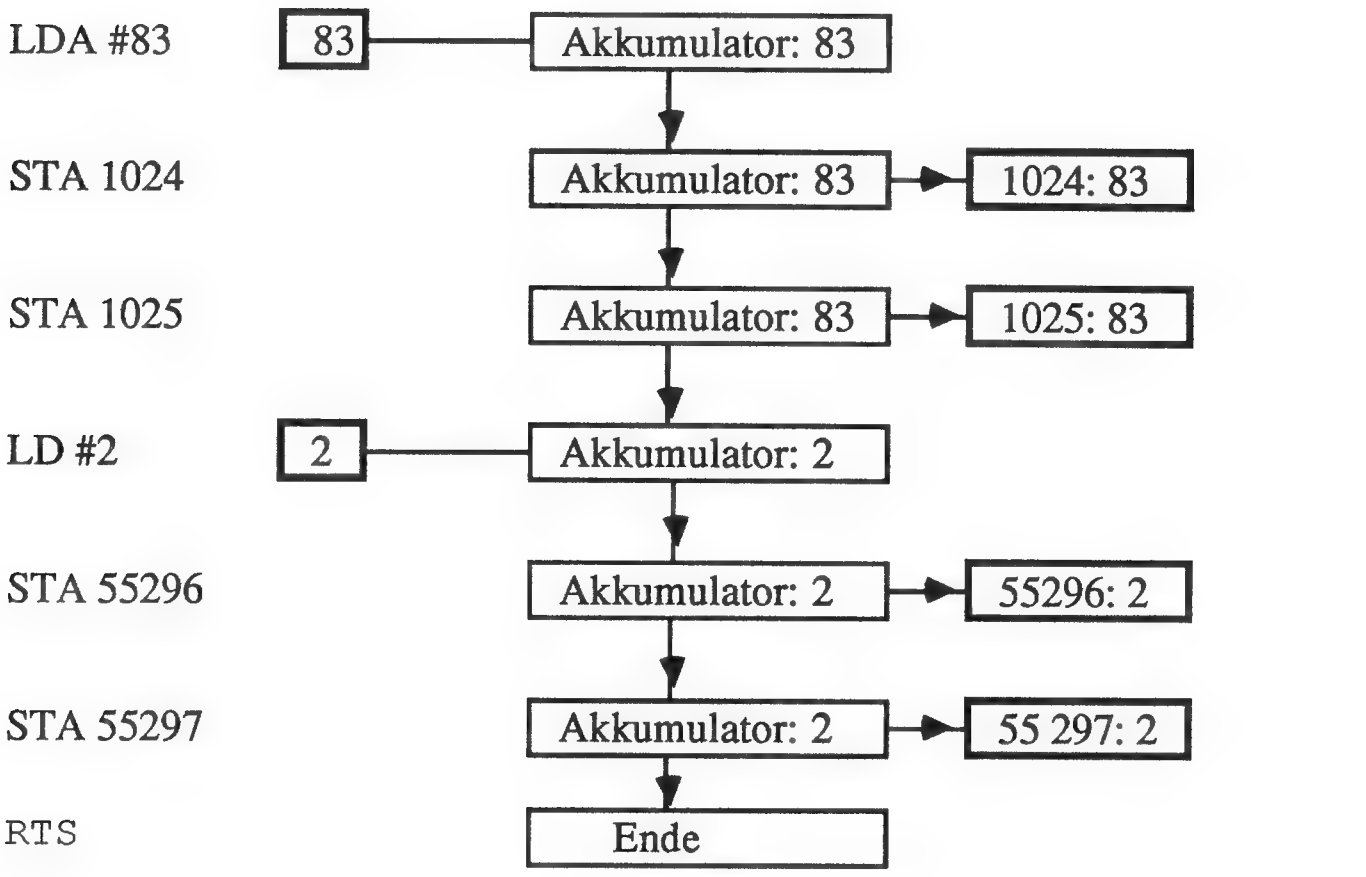


Abb. 1: Diagramm des Datentransfers

Aufgabe 3.1 Die Bildschirmcodes der Buchstaben A bis C können durch Rechnung ineinander übergeführt werden. Entweder beginnt man mit dem A, das den Code 1 besitzt, und errechnet jeweils

durch Addition von eins den nächsten Buchstaben, oder man beginnt mit dem C und subtrahiert jeweils eins. Die drei Buchstaben sollen oben links auf dem Bildschirm erscheinen. Dies entspricht den Adressen 1024 bis 1026 und im Farbspeicher den Adressen 55296 bis 55298.

Zuerst die Lösung durch Additionen:

```
LDA    #1                ;AKKU. MIT CODE VON "A" LADEN
STA    1024              ;DIESEN CODE IN DEN BILDSPEICHER
CLC                      ;DIE ADDITION VORBEREITEN
ADC    #1                ;DEN NEUEN CODE ERRECHNEN
STA    1025              ;DIESEN CODE IN DEN BILDSPEICHER
CLC                      ;DIE ADDITION VORBEREITEN
ADC    #1                ;DEN NEUEN CODE ERRECHNEN
STA    1026              ;DIESEN CODE IN DEN BILDSPEICHER
LDA    #6                ;AKKU. MIT CODE VON BLAU LADEN
STA    55296             ;DIESE FARBE IN DEN FARBSPEICHER
STA    55297             ;NOCHMALS IN DEN FARBSPEICHER
STA    55298             ;NOCHMALS IN DEN FARBSPEICHER
RTS                      ;DAS PROGRAMM VERLASSEN
```

Jetzt die Lösung des Problems über Subtraktionen:

```
LDA    #3                ;AKKU. MIT CODE VON "C" LADEN
STA    1026              ;DIESEN CODE IN DEN BILDSPEICHER
SEC                      ;DIE SUBTRAKTION VORBEREITEN
SBC    #1                ;DEN NEUEN CODE ERRECHNEN
STA    1025              ;DIESEN CODE IN DEN BILDSPEICHER
SEC                      ;DIE SUBTRAKTION VORBEREITEN
SBC    #1                ;DEN NEUEN CODE ERRECHNEN
STA    1024              ;DIESEN CODE IN DEN BILDSPEICHER
LDA    #6                ;AKKU. MIT CODE VON BLAU LADEN
STA    55296             ;DIESE FARBE IN DEN FARBSPEICHER
STA    55297             ;NOCHMALS IN DEN FARBSPEICHER
STA    55298             ;NOCHMALS IN DEN FARBSPEICHER
RTS                      ;DAS PROGRAMM VERLASSEN
```

Aufgabe 3.2 Der Code jedes einzelnen Buchstabens muß nacheinander in das X-Register geladen und in den Bildschirmspeicher geschrieben werden. Für die Farbinformation wird das Y-Register geladen und in den Farbspeicher geschrieben. In der Musterlösung wird der Text "SYBEX" in grüner Farbe auf dem Bildschirm ausgegeben.

```
LDX    #19               ;X-REG. MIT CODE VON "S" LADEN
STX    1024              ;UND IN DEN BILDSCHIRMSPEICHER
LDX    #25               ;X-REG. MIT CODE VON "Y" LADEN
STX    1025              ;UND IN DEN BILDSCHIRMSPEICHER
LDX    #2                ;X-REG. MIT CODE VON "B" LADEN
STX    1026              ;UND IN DEN BILDSCHIRMSPEICHER
```

```
LDX    #5                ;X-REG. MIT DEM CODE "E" LADEN
STX    1027              ;UND IN DEN BILDSCHIRMSPEICHER
LDX    #24               ;X-REG. MIT DEM CODE "X" LADEN
STX    1028              ;UND IN DEN BILDSCHIRMSPEICHER
LDY    #5                ;Y-REG. MIT CODE VON GRÜN LADEN
STY    55296             ;DIESEN WERT MEHRMALS IN DEN
STY    55297             ;FARBSPEICHER SCHREIBEN
STY    55298
STY    55299
STY    55300
RTS                      ;DAS PROGRAMM VERLASSEN
```

Aufgabe 3.3 Die Adresse der oberen linken Ecke ist 1024, der rechten oberen Ecke 1024+39=1063. Die linke untere Ecke liegt 24 Zeilen tiefer. Jede Zeile umfaßt 40 Zeichen, womit sich die Adresse 1024 + 24 * 40 = 1984 für die linke untere Ecke ergibt. Die rechte untere Ecke wird entsprechend errechnet: 1984 + 39 = 2023. Für den Farbspeicher ergeben sich nach diesen Berechnungen die Werte 55296, 55335, 56256 und 56295. Der Bildschirmcode eines Dezimalpunktes kann durch Rechnung in den Code der Farbe Weiß übergeführt werden. Zuerst eine Musterlösung unter Verwendung einer Subtraktion:

```
LDY    #46               ;Y-REG. MIT CODE VON "." LADEN
STY    1024              ;IN DIE LINKE OBERE ECKE
STY    1063              ;IN DIE RECHTE OBERE ECKE
STY    1984              ;IN DIE LINKE UNTERE ECKE
STY    2023              ;IN DIE RECHTE UNTERE ECKE
TYA                      ;Y-REG. IN DEN AKKU. KOPIEREN
SEC                      ;DIE SUBTRAKTION VORBEREITEN
SBC    #45               ;DEN FARBCODE WEISS ERRECHNEN
TAY                      ;DAS ERGEBNIS IN DAS Y-REG.
STY    55296             ;IN DIE LINKE OBERE ECKE
STY    55335             ;IN DIE RECHTE OBERE ECKE
STY    56256             ;IN DIE LINKE UNTERE ECKE
STY    56295             ;IN DIE RECHTE UNTERE ECKE
RTS                      ;DAS PROGRAMM VERLASSEN
```

Um das Problem durch eine Addition zu lösen, muß zuerst der Farbcode in das Y-Register geladen und in den Farbspeicher geschrieben werden. Durch eine Addition kann dann der Bildschirmcode eines Dezimalpunktes errechnet werden.

```
LDY    #1                ;Y-REG. MIT FARBCODE VON WEISS LADEN
STY    55296             ;IN DIE LINKE OBERE ECKE
STY    55335             ;IN DIE RECHTE OBERE ECKE
STY    56256             ;IN DIE LINKE UNTERE ECKE
STY    56295             ;IN DIE RECHTE UNTERE ECKE
TYA                      ;Y-REG. IN DEN AKKU. KOPIEREN
CLC                      ;DIE ADDITION VORBEREITEN
ADC    #45               ;DEN ZEICHENCODE VON "." ERRECHNEN
```

```
TAY                     ;DAS ERGEBNIS IN DAS Y-REG.
STY    1024             ;IN DIE LINKE OBERE ECKE
STY    1063             ;IN DIE RECHTE OBERE ECKE
STY    1984             ;IN DIE LINKE UNTERE ECKE
STY    2023             ;IN DIE RECHTE UNTERE ECKE
RTS                     ;DAS PROGRAMM VERLASSEN
```

Aufgabe 3.4 Die Verwendung des X-Registers statt des Y-Registers ist leicht zu erreichen. Die Befehle lauten z.B. einfach LDX statt LDY oder STX statt STY. Hier sehen Sie auch den logischen Aufbau der Mnemoniks. Um das Register zu wechseln, muß nur ein Buchstabe im Befehlswort verändert werden. Um einen Dekrementier-Befehl zum Einsatz zu bringen, muß mit dem höheren Wert begonnen werden und der niedrigere errechnet werden.

Das Programm muß also zuerst den Farbwert, die größere Zahl und dann den Bildschirmcode, die niedrigere Zahl, in den Speicher schreiben.

```
LDX    #1               ;X-REG. MIT FARBCODE WEISS LADEN
STX    55296            ;DIESEN WERT IN DEN FARBSPEICHER
DEX                     ;DEN BILDSCHIRMCODE ERRECHNEN
STX    1024             ;NEUEN WERT IN DEN BILDSPEICHER
RTS                     ;DAS PROGRAMM VERLASSEN
```

Aufgabe 3.5 Die Bildschirmcodes der Buchstaben A bis D können durch Inkrementieren bzw. Dekrementieren ineinander übergeführt werden. Zum Inkrementieren wird mit dem Buchstaben A begonnen, beim Dekrementieren mit dem Buchstaben D. Hier eine Lösung mit dem Inkrementier-Befehl und dem X-Register.

```
LDX    #1               ;CODE VON "A" INS X-REGISTER
STX    1024             ;WERT IN DEN BILDSCHIRMSPEICHER
INX                     ;CODE VON "B" ERRECHNEN
STX    1025             ;WERT IN DEN BILDSCHIRMSPEICHER
INX                     ;CODE VON "C" ERRECHNEN
STX    1026             ;WERT IN DEN BILDSCHIRMSPEICHER
INX                     ;CODE VON "D" ERRECHNEN
STX    1027             ;WERT IN DEN BILDSCHIRMSPEICHER
LDX    #8               ;FARBCODE VON ORANGE LADEN
STX    55296            ;UND IN DEN FARBSPEICHER
STX    55297            ;NOCHMALS IN DEN FARBSPEICHER
STX    55298            ;NOCHMALS IN DEN FARBSPEICHER
STX    55299            ;NOCHMALS IN DEN FARBSPEICHER
RTS                     ;DAS PROGRAMM VERLASSEN
```

Natürlich kann statt des X-Registers auch das Y-Register Verwendung finden. Werden zusätzlich noch Dekrementier-Befehle herangezogen, sieht dies wie folgt aus:

```
        LDY   #4              ;CODE VON "D" INS Y-REGISTER
        STY   1027            ;WERT IN DEN BILDSCHIRMSPEICHER
        DEY                   ;CODE VON "C" ERRECHNEN
        STY   1026            ;WERT IN DEN BILDSCHIRMSPEICHER
        DEY                   ;CODE VON "B" ERRECHNEN
        STY   1025            ;WERT IN DEN BILDSCHIRMSPEICHER
        DEY                   ;CODE VON "A" ERRECHNEN
        STY   1024            ;WERT IN DEN BILDSCHIRMSPEICHER
        LDY   #8              ;FARBCODE VON ORANGE LADEN
        STY   55296           ;UND IN DEN FARBSPEICHER
        STY   55297           ;NOCHMALS IN DEN FARBSPEICHER
        STY   55298           ;NOCHMALS IN DEN FARBSPEICHER
        STY   55299           ;NOCHMALS IN DEN FARBSPEICHER
        RTS                   ;DAS PROGRAMM VERLASSEN
```

Aufgabe 4.1 Das Unterprogramm zum Schreiben der Farbinformation soll ab der Adresse $1400 im Speicher stehen.

```
        *=    $1300           ;START DES HAUPTPROGRAMMS
        LDA   #3              ;CODE EINES "C" IN DEN AKKU.
        STA   1024            ;IN DEN BILDSCHIRMSPEICHER
        JSR   FARBE           ;DAS UNTERPROGRAMM AUFRUFEN
        RTS                   ;DAS PROGRAMM VERLASSEN
;
        *=    $1400           ;START DES UNTERPROGRAMMS
FARBE   LDA   #12             ;AKKU. MIT FARBCODE GRAU LADEN
        STA   55296           ;IN DEN FARBSPEICHER
        RTS                   ;DAS UNTERPROGRAMM VERLASSEN
```

Aufgabe 4.2 Der ASCII-Code jedes einzelnen Zeichens muß in den Akkumulator geladen werden und mit der Routine BSOUT ausgegeben werden. Es ist sinnvoll, die Startadresse der Routine BSOUT in einem Label zu definieren. Auch von der Möglichkeit, in mathematischen Ausdrücken direkt den ASCII-Wert eines Zeichens einzusetzen, wird ausgiebig Gebrauch gemacht.

```
BSOUT   =   $FFD2
        LDA   #"S"            ;ASCII-CODE VON "S" LADEN
        JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
        LDA   #"Y"            ;ASCII-CODE VON "Y" LADEN
        JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
        LDA   #"B"            ;ASCII-CODE VON "B" LADEN
        JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
        LDA   #"E"            ;ASCII-CODE VON "E" LADEN
        JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
        LDA   #"X"            ;ASCII-CODE VON "X" LADEN
        JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
        RTS                   ;DAS PROGRAMM VERLASSEN
```

Aufgabe 4.3 Zur Ausgabe der fünf Kommata muß eine Schleife verwendet werden, die von 5 nach 1 abwärts zählt. Entsprechend muß zur Ausgabe der Dezimalpunkte eine Schleife von 8 nach 1 herabzählen. Damit die beiden Ausgaben in verschiedenen Zeilen erscheinen, muß nach der Ausgabe der Kommata ein Wagenrücklauf über die Routine BSOUT ausgegeben werden. Das gesamte Programm sieht wie folgt aus:

```
    BSOUT  =    $FFD2
           LDX  #5              ;X-REG.= SCHLEIFENZAEHLER = 5
           LDA  #","            ;AKKU. MIT CODE VON "," LADEN
SCHLEIFE1  JSR  BSOUT           ;INHALT VON AKKU. AUSGEBEN
           DEX                  ;SCHLEIFENZAEHLER MINUS EINS
           BNE  SCHLEIFE1       ;NULL (ENDE) ?, NEIN =>
;
           LDA  #13             ;AKKU. MIT CODE VON CR LADEN
           JSR  BSOUT           ;DIESEN CODE AUSGEBEN
;
           LDY  #8              ;Y-REG.=SCHLEIFENZAEHLER=8
           LDA  #"."            ;AKKU. MIT CODE VON "." LADEN
SCHLEIFE2  JSR  BSOUT           ;INHALT VON AKKU. AUSGEBEN
           DEY                  ;SCHLEIFENZAEHLER MINUS EINS
           BNE  SCHLEIFE2       ;NULL (ENDE)?, NEIN =>
           RTS                  ;DAS PROGRAMM VERLASSEN
```

Natürlich kann auch das X-Register zur Ausgabe der Punkte bzw. das Y-Register zur Ausgabe der Kommata verwendet werden.

Aufgabe 4.4 Um einen Kreis von oben nach unten zu bewegen, muß der Kreis gezeichnet werden (Ausgabe von "●"), dann wieder gelöscht (Ausgabe von "DEL") und der Cursor eine Zeile tiefer bewegt werden (Ausgabe von "CRSR.DW."). Die einzelnen Zeichen bzw. Steuercodes werden der Reihe nach über die Routine BSOUT ausgegeben.

```
ZAEHLER    =    5000
BSOUT      =    $FFD2
;
           LDX  #15             ;X-REG, DIENT ALS SCHLEIFENZAEHLER
SCHLEIFE   LDA  #20             ;CODE VON "DEL" IN DEN AKKU.
           JSR  BSOUT           ;DEN LETZTEN KREIS LOESCHEN
           LDA  #17             ;CODE VON CRSR.-ABWAERTS IN DEN AKKU.
           JSR  BSOUT           ;DEN CODE AUSGEBEN
           LDA  #113            ;CODE EINES KREISES IN DEN AKKU.
           JSR  BSOUT           ;DEN AKKU.INHALT AUSGEBEN
;
           LDY  #100            ;AEUSSEREN SCHLEIFENZAEHLER MIT
           STY  ZAEHLER         ;STARTWERT LADEN
WARTEN1    LDY  #255            ;INNERE SCHLEIFE MIT STARTWERT LADEN
WARTEN2    DEY                  ;DEN INNEREN ZAEHLER VERMINDERN
```

```
        BNE   WARTEN2          ;INNERER ZAEHLER=0 ?,NEIN =>
        DEC   ZAEHLER          ;DEN AEUSSEREN ZAEHLER VERMINDERN
        BNE   WARTEN1          ;AUESSERER ZAEHLER=0 ?,NEIN =>

        DEX                    ;SCHLEIFENZAEHLER UM EINS VERMINDERN
        BNE   SCHLEIFE         ;SCHLEIFENZAEHLER=0 ?,NEIN =>
        RTS                    ;DAS PROGRAMM VERLASSEN
```

Durch Verändern der Warteschleife kann die Bewegungsgeschwindigkeit des Kreises variiert werden. Wird z.B. der Schleifenzähler ZAEHLER nur mit dem Wert 20 geladen, bewegt sich der Kreis beträchtlich schneller. Umgekehrt wird dieser langsamer, wenn der Zähler mit dem Wert 250 geladen wird.

Aufgabe 4.5 Der Schleifenzähler durchläuft die Werte 246 bis 255, also genau die ASCII-Codes der Zeichen, die ausgegeben werden sollen. Der Wert des Schleifenzählers muß einfach in den Akkumulator kopiert und dieser dann mit BSOUT ausgegeben werden. Den Kopiervorgang übernimnmt der Transfer-Befehl TYA.

```
BSOUT     =     $FFD2
          LDY   #246             ;Y-REG.=SCHLEIFENZAEHLER=246
SCHLEIFE  TYA                    ;SCHLEIFENZAEHLER NACH AKKUMULATOR
          JSR   BSOUT            ;DIESEN WERT AUSGEBEN
          INY                    ;SCHLEIFENZAEHLER PLUS EINS
          BNE   SCHLEIFE         ;NULL (ENDE)?, NEIN =>
          RTS                    ;DAS PROGRAMM VERLASSEN
```

Aufgabe 4.6 Der Schleifenzähler muß zur rückwärtigen Ausgabe des Alphabets von 90 abwärts nach 64 zählen. Der Schleifenzähler wird also nicht inkrementiert, sondern dekrementiert. Am genannten Endwert erkennen Sie auch, daß der Zähler wieder um eins weiter zählt, als eigentlich ASCII-Codes ausgegeben werden.

```
BSOUT     =     $FFD2
          LDX   #90              ;STARTWERT=90=ASCII-CODE VON "Z"
SCHLEIFE  TXA                    ;SCHLEIFENZAEHLER IN AKKUMULATOR
          JSR   BSOUT            ;DAS ZEICHEN AUSGEBEN
          DEX                    ;SCHLEIFENZAEHLER MINUS EINS
          CPX   #64              ;MIT DEM ENDWERT VERGLEICHEN
          BNE   SCHLEIFE         ;IDENTISCH (ENDE)?, NEIN =>
          RTS                    ;DAS PROGRAMM VERLASSEN
```

Aufgabe 4.7 Damit die Steuercodes nicht erscheinen, müssen zwei Schleifen jeweils die Codes von 32-127 bzw. von 160-255 ausgeben. Das Programm besteht also nur aus zwei einfachen Schleifen.

```
BSOUT      =      $FFD2
           LDX    #32              ;STARTWERT=32
SCHLEIFE1  TXA                     ;CODE IN DEN AKKUMULATOR
           JSR    BSOUT            ;DEN CODE AUSGEBEN
           INX                     ;DEN SCHLEIFENZAEHLER ERHOEHEN
           CPX    #128             ;MIT ENDWERT VERGLEICHEN
           BNE    SCHLEIFE1        ;IDENTISCH (ENDE)?, NEIN =>
;
           LDX    #160             ;STARTWERT=160
SCHLEIFE2  TXA                     ;CODE IN DEN AKKUMULATOR
           JSR    BSOUT            ;DEN CODE AUSGEBEN
           INX                     ;DEN SCHLEIFENZAEHLER ERHOEHEN
           BNE    SCHLEIFE2        ;NULL (ENDE)?, NEIN =>
           RTS                     ;DAS PROGRAMM VERLASSEN
```

Da der Endwert der zweiten Schleife 255 bzw. nach nochmaligem Inkrementieren 0 entspricht, kann die Abfrage des Endwertes direkt mit dem BNE-Befehl erfolgen.

Aufgabe 4.8 Die Assembleranweisung .TEXT erlaubt es, Zeichenfolgen gemischt mit Zahlenwerten bzw. Steuercodes in den Objektcode einzubauen. Um jetzt die gesamte Adresse auszugeben, wird zwischen den einzelnen Bestandteilen ein Wagenrücklauf eingefügt. Das Programm sieht dann wie folgt aus:

```
BSOUT      =     $FFD2
           LDX    #0               ;INDEX = 0
SCHLEIFE   LDA    ADRESSE,X        ;ZEICHEN AUS DER TABELLE LESEN
           BEQ    ENDE             ;DAS ENDEZEICHEN ?,JA =>
           JSR    BSOUT            ;DAS ZEICHEN AUSGEBEN
           INX                     ;DEN INDEX ERHOEHEN
           JMP    SCHLEIFE         ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                     ;DAS PROGRAMM WIEDER VERLASSEN
;
ADRESSE    .TEXT "SYBEX-VERLAG GMBH",13 ;NAME + CR
           .TEXT "VOGELSANGERWEG 111",13 ;STRASSE + CR
           .TEXT "4000 DÜSSELDORF 30",13,0 ;ORT + CR + ENDE
```

Aufgabe 4.9 Durch Prüfen des Schleifenzählers, kann bei Erreichen eines Maximalwertes die Schleife abgebrochen werden.

```
BSOUT      =     $FFD2
           LDX    #0               ;INDEX = 0
SCHLEIFE   LDA    NAME,X           ;ZEICHEN AUS DER TABELLE LESEN
           BEQ    ENDE             ;DAS ENDEZEICHEN ?,JA =>
           JSR    BSOUT            ;DAS ZEICHEN AUSGEBEN
           INX                     ;DEN INDEX ERHOEHEN
           CPX    #200             ;MIT MAXIMALWERT VERGLEICHEN
           BNE    SCHLEIFE         ;IDENTISCH (ENDE)?, NEIN =>
ENDE       RTS                     ;DAS PROGRAMM WIEDER VERLASSEN
;
NAME       .TEXT "SYBEX-VERLAG",0   ;AUSGABETEXT
```

Aufgabe 4.10 Bei diesem Programm wird jedes Zeichen nacheinander mit jedem Vokal verglichen. Handelt es sich bei dem Zeichen um einen Vokal, wird das Zeichen ausgegeben. War es ein Konsonant oder ein sonstiges Zeichen, "fällt" das Zeichen sozusagen durch die Abfragen hindurch. Dort wird die Ausgabe einfach durch einen Sprungbefehl übergangen.

```
BSOUT      =     $FFD2
;
           LDX   #0              ;INDEX = 0
SCHLEIFE   LDA   NAME,X          ;ZEICHEN AUS DER TABELLE LESEN
           BEQ   ENDE            ;DAS ENDEZEICHEN ?,JA =>
           CMP   #"A"            ;VOKAL "A" ?
           BEQ   AUSGABE         ;JA => (AUSGABE)
           CMP   #"E"            ;VOKAL "E" ?
           BEQ   AUSGABE         ;JA => (AUSGABE)
           CMP   #"I"            ;VOKAL "I" ?
           BEQ   AUSGABE         ;JA => (AUSGABE)
           CMP   #"O"            ;VOKAL "O" ?
           BEQ   AUSGABE         ;JA => (AUSGABE)
           CMP   #"U"            ;VOKAL "U" ?
           BEQ   AUSGABE         ;JA => (AUSGABE)
           JMP   WEITER          ;BEI ALLEN KONSONATEN
AUSGABE    JSR   BSOUT           ;ZEICHEN AUSGEBEN
WEITER     INX                   ;DEN INDEX ERHOEHEN
           JMP   SCHLEIFE        ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                   ;DAS PROGRAMM VERLASSEN
;
NAME       .TEXT "SYBEX-VERLAG",0 ;AUSGABETEXT
```

Aufgabe 4.11 Zur Lösung dieses Problems wird jedes Zeichen mit der Endmarke verglichen und bei Übereinstimmung die Schleife beendet.

```
BSOUT      =     $FFD2
           LDX   #0              ;INDEX = NULL
SCHLEIFE   LDA   NAME,X          ;ZEICHEN AUS DER TABELLE LESEN
           CMP   #","            ;MIT DEM ENDZEICHEN VERGLEICHEN
           BEQ   ENDE            ;IDENTISCH (ENDE)?, JA =>
           JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
           INX                   ;DEN INDEX ERHOEHEN
           JMP   SCHLEIFE        ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                   ;DAS PROGRAMM WIEDER VERLASSEN
;
NAME       .TEXT "SYBEX-VERLAG,"  ;AUSGABETEXT
```

Aufgabe 5.1 Die Startadresse des Textes wird jetzt in zwei Speicherzellen der ersten 256 Bytes geschrieben. Um die Adresse in das LO-Byte und das HI-Byte zu spalten, werden die beiden Operatoren "<" und ">" eingesetzt. Der Assembler errechnet dann die entsprechenden Werte. Als Index muß bei der nach-indizierten,

indirekten Adressierung das Y-Register verwendet werden. Die Ausgabe sieht dann wie folgt aus:

```
ZEIGER     =     250
BSOUT      =     $FFD2
           LDA   #<TEXT              ;LO-BYTE DES TEXTES LADEN
           STA   ZEIGER              ;IN DEN ZEIGER SPEICHERN
           LDA   #>TEXT              ;HI-BYTE DES TEXTES LADEN
           STA   ZEIGER+1            ;IN DEN ZEIGER SPEICHERN
           LDY   #0                  ;INDEX = NULL
SCHLEIFE   LDA   (ZEIGER),Y          ;ZEICHEN AUS DER TABELLE LESEN
           BEQ   ENDE                ;NULL (ENDE)?, JA =>
           JSR   BSOUT               ;DAS ZEICHEN AUSGEBEN
           INY                       ;DEN INDEX ERHOEHEN
           JMP   SCHLEIFE            ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                       ;DAS  PROGRAMM  WIEDER  VERLASSEN
;
TEXT       .TEXT "SYBEX-VERLAG",0    ;AUSGABETEXT
```

Aufgabe 5.2 Zur Lösung dieses Problems werden innerhalb der Schleife Zeichen aus zwei Texten gelesen und nacheinander ausgegeben.

Beim Lesen der Zeichen wird gleichzeitig die Endemarke überprüft. Ist diese bei einer Abfrage erreicht, wird die Schleife abgebrochen. Damit ist auch automatisch die Textausgabe im jeweils anderen Text beendet.

```
BSOUT      =     $FFD2
           LDX   #0                  ;INDEX = NULL
SCHLEIFE   LDA   TEXT1,X             ;ZEICHEN AUS DEM 1. TEXT LESEN
           BEQ   ENDE                ;NULL (ENDE)?, JA =>
           JSR   BSOUT               ;DAS ZEICHEN AUSGEBEN
           LDA   TEXT2,X             ;ZEICHEN AUS DEM 2. TEXT LESEN
           BEQ   ENDE                ;NULL (ENDE)?, JA =>
           JSR   BSOUT               ;DAS ZEICHEN AUSGEBEN
           INX                       ;DEN INDEX ERHOEHEN
           JMP   SCHLEIFE            ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE       RTS                       ;DAS PROGRAMM WIEDER VERLASSEN
;
TEXT1      .TEXT "SYBEX",0           ;1. AUSGABETEXT
TEXT2      .TEXT "VERLAG",0          ;2. AUSGABETEXT
```

Aufgabe 6.1 Bei der Addition der niederwertigen Stellen kann ein Übertrag auftreten. Dieser wird jedoch von der nächsten Stelle weiterverarbeitet. Eine Anzeige dieses Übertrags ist also nicht besonders zweckdienlich. Eine entsprechende Ausgabe sollte am Ende der gesamten Addition erfolgen.

```
BSOUT      =     $FFD2
SUMLO1     =     250          ;LO-BYTE DES 1. SUMMANDEN
SUMHI1     =     251          ;HI-BYTE DES 1. SUMMANDEN
SUMLO2     =     252          ;LO-BYTE DES 2. SUMMANDEN
```

```
SUMHI2    =     253          ;HI-BYTE DES 2. SUMMANDEN
ERGLO     =     252          ;LO-BYTE DES ERGEBNISSES
ERGHI     =     253          ;HI-BYTE DES ERGEBNISSES
;
          LDA   SUMLO1       ;LO-BYTE DES 1. SUMMANDEN HOLEN
          CLC                ;DAS UEBERTRAGS-FLAG LOESCHEN
          ADC   SUMLO2       ;LO-BYTE DES 2. SUMM. ADDIEREN
          STA   ERGLO        ;ENDERGEBNIS LO-BYTE SPEICHERN
          LDA   SUMHI1       ;HI-BYTE DES 1. SUMMANDEN HOLEN
          ADC   SUMHI2       ;HI-BYTE DES 2. SUMM. ADDIEREN
          STA   ERGHI        ;ENDERGEBNIS HI-BYTE SPEICHERN
          BCC   ENDE         ;UEBERTRAG ?, NEIN =>
          LDA   #"C"         ;AKKU. MIT ASCII-CODE VON "C" LADEN
          JSR   BSOUT        ;DAS ZEICHEN AUSGEBEN
ENDE      RTS                ;DAS PROGRAMM VERLASSEN
```

Aufgabe 6.2 Ähnlich wie bei der Addition zweier 16-Bit-Zahlen werden auch bei der Subtraktion zuerst die niederwertigen Bytes und dann die höherwertigen Bytes verarbeitet.

```
BSOUT     =     $FFD2
MINLO     =     250          ;LO-BYTE DES MINUENDEN
MINHI     =     251          ;HI-BYTE DES MINUENDEN
SUBLO     =     252          ;LO-BYTE DES SUBTRAHENDEN
SUBHI     =     253          ;HI-BYTE DES SUBTRAHENDEN
ERGLO     =     252          ;LO-BYTE DES ERGEBNISSES
ERGHI     =     253          ;HI-BYTE DES ERGEBNISSES
;
          LDA   MINLO        ;LO-BYTE DES MINUENDEN HOLEN
          SEC                ;DAS UEBERTRAGS-FLAG SETZEN
          SBC   SUBLO        ;LO-BYTE DES SUBTRAHENDEN SUBTRAH.
          STA   ERGLO        ;LO-BYTE DES ENDERGEBNISSES SPEICHERN
          LDA   MINHI        ;HI-BYTE DES MINUENDEN HOLEN
          SBC   SUBHI        ;HI-BYTE DES SUBTRAHENDEN SUBTRAHIEREN
          STA   ERGHI        ;HI-BYTE DES ENDERGEBNISSES SPEICHERN
          BCC   ENDE         ;UEBERTRAG ?, NEIN =>
          LDA   #"C"         ;AKKU. MIT ASCII-CODE VON "C" LADEN
          JSR   BSOUT        ;DAS ZEICHEN AUSGEBEN
ENDE      RTS                ;DAS PROGRAMM VERLASSEN
```

Aufgabe 6.3 Am Ende der Subtraktion wird mit dem Branch-Befehl BVC geprüft, ob ein Überlauf eintrat. Durch die Ausgabe eines V wird dieser Fall angezeigt.

```
BSOUT     =     $FFD2
MINLO     =     250          ;LO-BYTE DES MINUENDEN
MINHI     =     251          ;HI-BYTE DES MINUENDEN
SUBLO     =     252          ;LO-BYTE DES SUBTRAHENDEN
SUBHI     =     253          ;HI-BYTE DES SUBTRAHENDEN
ERGLO     =     252          ;LO-BYTE DES ERGEBNISSES
ERGHI     =     253          ;HI-BYTE DES ERGEBNISSES
;
          LDA   MINLO        ;LO-BYTE DES MINUENDEN HOLEN
```

```
            SEC                 ;DAS UEBERTRAGS-FLAG SETZEN
            SBC   SUBLO         ;LO-BYTE DES SUBTRAHENDEN SUBTRAHIEREN
            STA   ERGLO         ;LO-BYTE DES ENDERGEBNISSES SPEICHERN
            LDA   MINHI         ;HI-BYTE DES MINUENDEN HOLEN
            SBC   SUBHI         ;HI-BYTE DES SUBTRAHENDEN SUBTRAHIEREN
            STA   ERGHI         ;HI-BYTE DES ENDERGEBNISSES SPEICHERN
            BCC   ENDE          ;UEBERTRAG ?, NEIN =>
            LDA   #"V"          ;AKKU. MIT ASCII-CODE VON "V" LADEN
            JSR   BSOUT         ;DAS ZEICHEN AUSGEBEN
ENDE        RTS                 ;DAS PROGRAMM VERLASSEN
```

Einige interessante Werte:

```
POKE 250,0:POKE 251,10:POKE 252,128:POKE 253,5 (2560-1408)
!GO...
?PEEK(252)+PEEK(253)*256
1152

POKE 250,100:POKE 251,138:POKE 252,0:POKE 253,20 (35428 - 5120)
!GO...
V
?PEEK(252)+PEEK(253)*256
30308
```

Im letzten Beispiel tritt ein Überlauf auf. 16-Bit-Zahlen dürfen Werte von 0 bis 65535 annehmen. In der Zweierkomplement-Darstellung werden Zahlen von 0 bis 32767 als positiv interpretiert und die Zahlen von 32768 bis 65535 als negativ. Mit 35428 – 5120 wurde in Wirklichkeit –30108 –5120 gerechnet.

Damit wurde die Grenze der negativen Zahlen überschritten, womit ein Überlauf auftrat (–30108 – 5120 = –35228, 35428 – 5120 = 30308 => positive Zahl).

Aufgabe 6.4 Nach der Addition der zwei Zahlen wird mit dem Befehl BVC das Auftreten eines Überlaufs geprüft.

```
BSOUT       =     $FFD2
SUM1        =     250           ;SPEICHERZELLE DES 1. SUMMANDEN
SUM2        =     251           ;SPEICHERZELLE DES 2. SUMMANDEN
ERG         =     252           ;SPEICHERZELLE DES EREGBNISSES
            LDA   SUM1          ;DEN 1. SUMMANDEN LADEN
            CLC                 ;DEN UEBERTRAG LOESCHEN
            ADC   SUM2          ;DEN 2. SUMMANDEN ADDIEREN
            STA   ERG           ;DAS ERGEBNIS MERKEN
            BVC   ENDE          ;UEBERLAUF ?, NEIN =>
            LDA   #"V"          ;AKKUMULATOR MIT CODE VON "V" LADEN
            JSR   BSOUT         ;DAS ZEICHEN AUSGEBEN
ENDE        RTS                 ;DAS PROGRAMM VERLASSEN
```

Einige Zahlenbeispiele:

```
POKE 250,10:POKE 251,78
!GO...
?PEEK(250)
88

POKE 250,70:POKE 251,80
!GO...
V
?PEEK(250)
150
```

Im letzten Beispiel wurden zwei positive Zahlen addiert. Das Ergebnis überschreitet jedoch den für positive Zahlen zulässigen Bereich. Es tritt ein Überlauf auf.

Aufgabe 6.5 Die Ausgabe des Textes "250 KLEINER 251" übernimmt eine Schleife, wie sie in Kapitel 4 des öfteren beschrieben wurde. Vor Ausführung der Schleife werden jedoch die Speicherzellen 250 und 251 miteinander verglichen. Nur wenn die Bedingung 250 kleiner 251 erfüllt ist, wird die Ausgabeschleife aufgerufen.

```
BSOUT     =    $FFD2
ZAHL1     =     250          ;SPEICHERZELLE DER 1. VERGLEICHSZAHL
ZAHL2     =     251          ;SPEICHERZELLE DER 2. VERGLEICHSZAHL
          LDA   250          ;1. ZAHL IN DEN AKKUMULATOR LADEN
          CMP   251          ;MIT DER ZWEITEN ZAHL VERGLEICHEN
          BCS   ENDE         ;250 >= 251 ?, JA =>
;
          LDX   #0           ;INDEX = 0
SCHLEIFE  LDA   TEXT,X       ;ZEICHEN AUS DER TABELLE LESEN
          BEQ   ENDE         ;DAS ENDEZEICHEN ?,JA =>
          JSR   BSOUT        ;DAS ZEICHEN AUSGEBEN
          INX                ;DEN INDEX ERHOEHEN
          BNE   SCHLEIFE     ;IDENTISCH (ENDE)?, NEIN =>
ENDE      RTS                ;DAS PROGRAMM WIEDER VERLASSEN
;
TEXT      .TEXT "250 KLEINER 251",0 ;AUSGABETEXT
```

Aufgabe 6.6 In der abgedruckten Lösung wird ein Index von eins und eine Schrittweite von drei verwandt. Es wird also das 2., 5., 8. usw. Zeichen ausgegeben. Um Komplikationen zu vermeiden, ist die Endmarke auf drei Nullen erweitert.

```
BSOUT     =    $FFD2
ZEIGER    =    250           ;ZEIGER IN DEN TEXT
;
          LDA  #<TEXT        ;LO-BYTE DES TEXTANFANGS
          STA  ZEIGER        ;ALS LO-BYTE DES ZEIGERS MERKEN
```

```
            LDA   #>TEXT          ;HI-BYTE DES TEXTANFANGS
            STA   ZEIGER+1        ;ALS HI-BYTE DES ZEIGERS MERKEN
            LDY   #1              ;YR=0 (FUER ADRESSIERUNG)
SCHLEIFE    LDA   (ZEIGER),Y      ;ZEICHEN AUS DEM TEXT HOLEN
            BEQ   ENDE            ;ENDMARKE ?,JA =>
            JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
            LDA   ZEIGER          ;LO-BYTE DES ZEIGERS IN AKKU.
            CLC                   ;C-FLAG LOESCHEN
            ADC   #3              ;DAS LO-BYTE ERHOEHEN
            STA   ZEIGER          ;DAS ERGEBNIS SPEICHERN
            BCC   WEITER          ;EIN UEBERTRAG?,NEIN =>
            INC   ZEIGER+1        ;HI-BYTE DES ZEIGERS ERHOEHEN
WEITER      JMP   SCHLEIFE        ;ZUM ANF. DER SCHLEIFE SPRINGEN
ENDE        RTS                   ;DAS PROGRAMM VERLASSEN
;
TEXT        .TEXT "SYBEX-VERLAG",0,0,0
```

Aufgabe 6.7 Das Programm besteht im Prinzip aus zwei Schleifen. In der ersten wird der Text vorwärts ausgegeben und in der zweiten rückwärts. Als Endmarke wird am einfachsten eine Null an beiden Enden des Textes verwendet. Die gesamte Texttabelle sieht zwar jetzt etwas kompliziert aus, aber der hier beschriebene Weg ist mit Sicherheit noch der kürzeste und einfachste.

```
BSOUT       =     $FFD2
ZEIGER      =     250             ;ZEIGER IN DEN TEXT
;
            LDA   #<TEXTANF       ;LO-BYTE DES TEXTANFANGS
            STA   ZEIGER          ;ALS LO-BYTE DES ZEIGERS MERKEN
            LDA   #>TEXTANF       ;HI-BYTE DES TEXTANFANGS
            STA   ZEIGER+1        ;ALS HI-BYTE DES ZEIGERS MERKEN
            LDY   #0              ;YR=0 (FUER ADRESSIERUNG)
SCHLEIFEV   LDA   (ZEIGER),Y      ;ZEICHEN AUS DEM TEXT HOLEN
            BEQ   RUECKW          ;ENDMARKE ?,JA =>
            JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
            INC   ZEIGER          ;DAS LO=BYTE ERHOEHEN
            BNE   WEITER          ;EIN UEBERTRAG?,NEIN =>
            INC   ZEIGER+1        ;HI-BYTE DES ZEIGERS ERHOEHEN
WEITER      JMP   SCHLEIFEV       ;ZUM ANF. DER SCHLEIFE SPRINGEN
;
RUECKW      LDA   #<TEXTEND       ;LO-BYTE DES TEXTENDES
            STA   ZEIGER          ;ALS LO-BYTE DES ZEIGERS MERKEN
            LDA   #>TEXTEND       ;HI-BYTE DES TEXTENDES
            STA   ZEIGER+1        ;ALS HI-BYTE DES ZEIGERS MERKEN
            LDY   #0              ;YR=0 (FUER ADRESSIERUNG)
SCHLEIFER   LDA   (ZEIGER),Y      ;ZEICHEN AUS DEM TEXT HOLEN
            BEQ   ENDE            ;ENDMARKE ?,JA =>
            JSR   BSOUT           ;DAS ZEICHEN AUSGEBEN
            LDA   ZEIGER          ;DAS LO-BYTE PRUEFEN
            BNE   LOBYTE          ;NULL ?, NEIN =>
            DEC   ZEIGER+1        ;DAS HI-BYTE VERMINDERN
LOBYTE      DEC   ZEIGER          ;DAS LO-BYTE VERMINDERN
            JMP   SCHLEIFER       ;ZUM ANF. DER SCHLEIFE SPRINGEN
```

```
ENDE      RTS                    ;DAS PROGRAMM VERLASSEN
;
          .BYTE 0                ;SCHLUSSNULL FUER RUECKWAERTS
TEXTANF   .TEXT "SYBEX-VERLA"    ;TEXT-ZWISCHENSTUECK
TEXTENDE  .TEXT "G",0            ;TEXTENDE UND SCHLUSSNULL
```

Aufgabe 6.8 Am schnellsten ist die Multiplikation mit drei durch drei Additionen zu bewerkstelligen. Im Programm sieht dies wie folgt aus:

```
ZAHL      =     250              ;ZU MULTIPLIZIERENDE ZAHL
ERG       =     251              ;SPEICHERZELLE DES ERGEBNISSES
          LDA   250              ;ZAHL IN DEN AKKUMULATOR LADEN
          CLC                    ;DEN UEBERTRAG LOESCHEN
          ADC   250              ;DIE ZAHL ADDIEREN (*2)
          CLC                    ;DEN UEBERTRAG LOESCHEN
          ADC   250              ;DIE ZAHL ADDIEREN (*3)
          STA   251              ;DAS ERGEBNIS MERKEN
          RTS                    ;DAS PROGRAMM BEENDEN
```

Für eine allgemeine Multiplikation könnten die einzelnen Additionen in eine Schleife gefaßt werden. Wie oft die Schleife durchlaufen wird, kann durch einen der beiden Faktoren bestimmt werden:

```
MKATOR    =     250              ;SPEICHERZELLE DES MULTIPLIKATORS
MKAND     =     251              ;SPEICHERZELLE DES MULTIPLIKANDEN
ERG       =     251              ;SPEICHERZELLE DES ERGEBNISSES
          LDA   #0               ;AKKUMULATOR MIT NULL LADEN
          LDX   MKAND            ;SCHLEIFENZAEHLER=MULTIPLIKAND
          BEQ   ENDE             ;MULTIPLIKAND=0 ?, JA =>
SCHLEIFE  CLC                    ;DEN UEBERTRAG LOESCHEN
          ADC   MKATOR           ;DEN MULTIPLIKATOR ADDIEREN
          DEX                    ;DEN SCHLEIFENZAEHLER VERMINDERN
          BNE   SCHLEIFE         ;ZUM SCHLEIFENANFANG SPRINGEN
ENDE      STA   ERG              ;DAS ERGEBNIS SPEICHERN
          RTS                    ;DAS PROGRAMM VERLASSSEN
```

Aufgabe 6.9 Bei der Addition des Multiplikators zum Ergebnis wird immer eine 8-Bit-Zahl zu einer 16-Bit-Zahl addiert. Zum HI-Byte wird also maximal ein Übertrag addiert. Dies kann zur Beschleunigung in einem Inkrementbefehl geschehen.

Das verbesserte Programm sieht dann wie folgt aus:

```
MKATOR    =     250              ;SPEICHERZELLE DES MULTIPLIKATORS
MKAND     =     251              ;SPEICHERZELLE DES MULTIPLIKANDEN
ERGLO     =     252              ;LO-BYTE DES ERGEBNISSES
ERGHI     =     253              ;HI-BYTE DES ERGEBNISSES
;
          LDA   #0               ;DAS ERGEBNIS MIT NULL VORBELEGEN
          STA   ERGLO
```

```
            STA   ERGHI
            LDX   #8              ;SCHLEIFENZAEHLER MIT 8 STARTEN
SCHLEIFE    ASL   ERGLO           ;DAS ERGEBNIS NACH LINKS SCHIEBEN
            ROL   ERGHI
            ASL   MKAND           ;BIT 7 VON MKAND INS C-FLAG SCHIEBEN
            BCC   WEITER          ;WAR ES EINE EINS?, NEIN =>
            LDA   ERGLO           ;LO-BYTE-ERGEBNIS IN DEN AKKU.
            CLC                   ;DAS UEBERTRAGSFLAG LOESCHEN
            ADC   MKATOR          ;DEN MULTIPLIKATOR ADDIEREN
            STA   ERGLO           ;DAS LO-BYTE SPEICHERN
            BCC   WEITER          ;EIN UEBERTRAG ?, NEIN =>
            INC   ERGHI           ;HI-BYTE DES ERGEBNISSES PLUS EINS
WEITER      DEX                   ;DEN SCHLEIFENZAEHLER VERMINDERN
            BNE   SCHLEIFE        ;NULL (ENDE) ERREICHT?,NEIN =>
            RTS                   ;DAS PROGRAMM VERLASSEN
```

Aufgabe 6.10 Eine Multiplikation mit 3 kann in folgende Einzelteile zerlegt werden, die leicht in ein Programm übernommen werden können:

$$3 * X = 2 * X + X$$

```
ZAHL        =     250             ;ZU MULTIPLIZIERENDE ZAHL
            LDA   ZAHL            ;DIE ZAHL IN DEN AKKUMULATOR LADEN
            ASL   A               ;DIESEN WERT *2
            CLC                   ;DEN UEBERTRAG LOESCHEN
            ADC   ZAHL            ;DIE ZAHL ADDIEREN (*3)
            STA   ZAHL            ;DAS ENDERGEBNIS MERKEN
            RTS                   ;DAS PROGRAMM VERLASSEN
```

Zur Multiplikation mit 10 wird die Multiplikation mit 5 aus Kapitel 6 einfach nochmals mit 2 multipliziert:

$$10 * X = (4 * X + X) * 2$$

```
ZAHL        =     250             ;ZU MULTIPLIZIERENDE ZAHLEN
            LDA   ZAHL            ;DIE ZAHL IN DEN AKKUMULATOR LADEN
            ASL   A               ;DEN WERT *2
            ASL   A               ;DEN WERT *2 (*4)
            CLC
            ADC   ZAHL            ;DIE ZAHL ADDIEREN (*5)
            ASL   A               ;DEN WERT *2 (*10)
            STA   ZAHL            ;DAS ERGEBNIS MERKEN
            RTS                   ;DAS PROGRAMM VERLASSEN
```

Aufgabe 6.11 Zu Beginn der gesamten Divisionsroutine muß der Divisor in den Akkumulator geladen werden. Durch einen Branch-Befehl wird festgestellt, ob dieser null ist. Gegebenenfalls erfolgt dann eine Ausgabe.

```
BSOUT     =     $FFD2
DIVDEND   =     250          ;SPEICHERZELLE DES DIVIDENDEN
DIVISOR   =     251          ;SPEICHERZELLE DES DIVISORS
ERG       =     252          ;SPEICHERZELLE DES ERGEBNISSES
;
          LDA   DIVISOR      ;DEN DIVISOR IN DEN AKKU. LADEN
          BEQ   FEHLER       ;NULL (FEHLER)?, JA=>
          LDA   #0           ;DAS ERGEBNIS MIT NULL BELEGEN
          STA   ERG
          LDX   #0           ;DEN ZAEHLER MIT DEM WERT 0 STARTEN
SCHIEBEN  INX                ;DEN ZAEHLER UM EINS ERHOEHEN
          LDA   DIVISOR      ;DEN DIVISOR IN DEN AKKU. LADEN
          BMI   SCHLEIFE     ;DIVISOR AN ZAHLGRENZE ?, JA =>
          ASL   DIVISOR      ;DEN DIVISOR NACH LINKS SCHIEBEN
          JMP   SCHIEBEN     ;ZUM ANFANG DER SCHIEBESCHLEIFE
;
SCHLEIFE  ASL   ERG          ;DAS ERGEBNIS NACH LINKS SCHIEBEN
          LDA   DIVIDEND     ;DEN DIVIDENDEN IN AKKUMULATOR HOLEN
          CMP   DIVISOR      ;DIVIDEND MIT DIVISOR VERGLEICHEN
          BCC   WEITER       ;DIVIDEND<DIVISOR?, JA =>
          SEC                ;DAS UEBERTRAGS-FLAG SETZEN
          SBC   DIVISOR      ;ENDGUELTIGE SUBTRAK. DURCHFUEHREN
          STA   DIVIDEND     ;DAS SUB.-ERGEBNIS SPEICHERN
          INC   ERG          ;ENDERGEBNIS+1
WEITER    LSR   DIVISOR      ;DIVISOR NACH RECHTS SCHIEBEN
          DEX                ;DEN ZAEHLER UM EINS VERMINDERN
          BNE   SCHLEIFE     ;NULL (ENDE)?, NEIN =>
          RTS                ;DAS PROGRAMM VERLASSEN
;
FEHLER    LDA   #"E"         ;AKKUMULATOR MIT "E" LADEN
          JSR   BSOUT        ;DIESES ZEICHEN AUSGEBEN
          RTS                ;DAS PROGRAMM VERLASSEN
```

In der Musterlösung sehen Sie auch, daß ein Programm mehrere Enden haben kann. Ist der Divisor ungleich null, wird der Divisionszweig ausgeführt und das Programm auch dort mit RTS verlassen. Ist der Divisor null, wird ein Fehler angezeigt und das Programm in diesem Zweig verlassen.

Aufgabe 6.12 Vor der Ausgabe der obersten Stelle wird durch einen Vergleich festgestellt, ob es sich bei dieser Ziffer um eine Null handelt. Ist dies der Fall, wird die Ausgabe der Stelle übersprungen. Eine Führungsnull erscheint nicht mehr.

```
BSOUT     =     $FFD2
SUM1      =     250          ;1. SUMMAND
SUM2      =     251          ;2. SUMMAND
ERG       =     251          ;ERGEBNIS
;
          SED                ;DEZIMAL-MODUS EINSCHALTEN
          LDA   SUM1         ;DEN 1. SUMMANDEN IN AKKU. LADEN
          CLC                ;DAS UEBERTRAGS-FLAG LOESCHEN
          ADC   SUM2         ;DEN 2. SUMMANDEN ADDIEREN
          STA   ERG          ;DAS ERGEBNIS SPEICHERN
```

```
        CLD                     ;DEN DEZIMAL-MODUS AUSSCHALTEN
;
        LSR A                   ;DIE OBERE STELLE VERSCHIEBEN
        LSR A
        LSR A
        LSR A
        CLC                     ;DEN UEBERTRAG LOESCHEN
        ADC     #48             ;DEN ASCII-CODE ERRECHNEN
        CMP     #"0"            ;MIT EINER NULL VERGLEICHEN
        BEQ     WEITER          ;IDENTISCH ?, JA =>
        JSR     BSOUT           ;UND DIE ZIFFER AUSGEBEN
WEITER  LDA     ERG             ;DAS ERGEBNIS IN DEN AKKU. LADEN
        AND     #15             ;DIE UNTERE STELLE ISOLIEREN
        CLC                     ;DEN UEBERTRAG LOESCHEN
        ADC     #48             ;DEN ASCII-CODE ERRECHNEN
        JMP     BSOUT           ;UND ZIFFER AUSGEBEN+PROG. VERLASSEN
```

Aufgabe 6.13 Die Subtraktion im Dezimal-Modus erfolgt in der gleichen Weise wie bei der Addition. Es wird der Dezimal-Modus eingeschaltet, die Subtraktion wie gewohnt durchgeführt und dann der Dezimal-Modus wieder abgeschaltet. Die Ausgabe ist identisch mit der Ausgabe bei der Addition. In der Musterlösung wird wieder eine Führungsnull unterdrückt.

```
BSOUT   =       $FFD2
MIN     =       250             ;MINUEND
SUB     =       251             ;SUBTRAHEND
ERG     =       251             ;ERGEBNIS
;
        SED                     ;DEZIMAL-MODUS EINSCHALTEN
        LDA     MIN             ;DEN MINUEND IN AKKU. LADEN
        SEC                     ;DAS UEBERTRAGS-FLAG SETZEN
        SBC     SUB             ;DEN SUBTRAHEND SUBTRAHIEREN
        STA     ERG             ;DAS ERGEBNIS SPEICHERN
        CLD                     ;DEN DEZIMAL-MODUS AUSSCHALTEN
;
        LSR     A               ;DIE OBERE STELLE VERSCHIEBEN
        LSR     A
        LSR     A
        LSR     A
        CLC                     ;DEN UEBERTRAG LOESCHEN
        ADC     #48             ;DEN ASCII-CODE ERRECHNEN
        CMP     #"0"            ;MIT EINER NULL VERGLEICHEN
        BEQ     WEITER          ;IDENTISCH ?, JA =>
        JSR     BSOUT           ;UND DIE ZIFFER AUSGEBEN
WEITER  LDA     ERG             ;DAS ERGEBNIS IN DEN AKKU. LADEN
        AND     #15             ;DIE UNTERE STELLE ISOLIEREN
        CLC                     ;DEN UEBERTRAG LOESCHEN
        ADC     #48             ;DEN ASCII-CODE ERRECHNEN
        JMP     BSOUT           ;UND ZIFFER AUSGEBEN+PROG. VERLASSEN
```

Aufgabe 7.1 Zuerst werden die Operationen in der vorgegebenen Reihenfolge ausgeführt:

```
        00110101    (53)
UND     10101100    (172)
        --------
        00100100
ODER    01000001    (65)
        --------
        01100101
EXOR    10000101    (133)
        --------
        11100000
UND     01001101    (77)
        --------
        01000000    = 64
```

Nochmals die gleiche Rechnung, jedoch in anderer Reihenfolge:

```
        00110101    (53)
ODER    01000001    (65)
        --------
        01110101
UND     01001101    (77)
        --------
        01000101
UND     10101100    (172)
        --------
        00000100
EXOR    10000101    (133)
        --------
        10000001    = 129
```

Wie zu erkennen ist, darf die Reihenfolge der einzelnen Operationen nicht vertauscht werden.

Aufgabe 7.2 Vor der Addition wird der Subtrahend negiert. Die Addition entspricht dann einer Subtraktion (A – B = A + (– B)). Im Programm sehen beide Schritte zusammen wie folgt aus:

```
MIN     =     250    ;SPEICHERZELLE DES MINUENDEN
SUB     =     251    ;SPEICHERZELLE DES SUBTRAHENDEN
ERG     =     252    ;EREGBNIS DER RECHNUNG
        LDA   SUB    ;DEN SUBTRAHENDEN IN DEN AKKU. LADEN
```

```
        EOR   #255          ;DAS EINERKOMPLEMENT BILDEN
        CLC                 ;DEN UEBERTRAG LOESCHEN
        ADC   #1            ;EINS ADDIEREN (=ZWEIERKOMPLEMENT)
        STA   SUB           ;DAS ERGEBNIS ALS SUBTRAHEND MERKEN

        LDA   MIN           ;DEN MINUENDEN IN DEN AKKU. LADEN
        CLC                 ;DEN UEBERTRAG LOESCHEN
        ADC   SUB           ;DEN SUBTRAHENDEN ADDIEREN
        STA   ERG           ;DAS ERGEBNIS MERKEN
        RTS                 ;DAS PROGRAMM VERLASSEN
```

Aufgabe 8.1 Das Statusregister wird auf den Stapel gespeichert. Der Stapelwert, jetzt der Inhalt des Statusregisters, wird von dort in den Akkumulator geladen und kann in eine Speicherzelle geschrieben werden. Dies ist der einzige Weg, den Inhalt des Statusregisters als Ganzes zu lesen.

```
SPEICHER =    250           ;SPEICHERZELLE DES ERGEBNISSES
        PHP                 ;DAS STATUSREGISTER ABLEGEN
        PLA                 ;DEN WERT IN DEN AKKUMULATOR LADEN
        STA   SPEICHER      ;DEN WERT IN DEN SPEICHER SCHREIBEN
        RTS                 ;DAS PROGRAMM VERLASSEN
```

Anhang B

Adressierungsarten

Adressierungsart	Eingabeformat

implizit *leer*

Bei Befehlen mit dieser Adressierart handelt es sich immer um ein Byte lange Befehle. Diese lassen andere Adressierungen nicht zu. Das Befehlswort selbst enthält die Information über Quelle und Ziel der Daten.

Akkumulator *A oder leer*

Mit dieser Adessierungsart wird bei den Schiebe- und Rotationsbefehlen angegeben, daß der Inhalt des Akkumulators beeinflußt werden soll. Die Befehle haben die Länge eines Bytes.

unmittelbar *# data* (8-Bit-Wert)

Die Bezeichnung lautet bei dieser Adressierung "unmittelbar", weil der Zahlenwert im Programm direkt aufgeführt wird. Akkumulator, X- und Y-Register werden über diese Adressierart mit einem bestimmten Wert geladen. Die Befehle sind zwei Bytes lang.

Zero-Page *addr* (8-Bit-Wert)

Bei dieser Adressierung wird eine Speicherzelle in den ersten 256 Bytes angesprochen. Diese Adressierung ist kürzer, nur zwei Bytes, und schneller als eine entsprechende absolute Adressierung.

absolut *addr 16* (16-Bit-Wert)

Hier wird im Operanden die Adresse der gewünschten Speicherzelle angegeben. Deren Inhalt wird dann gelesen, beschrieben oder verändert. Befehle mit dieser Adressierung sind drei Bytes lang.

Adressierungsart	Eingabeformat	
indiziert (X)	*addr,X*	(8-Bit-Wert)

Bei dieser Adressierungsart wird die eigentliche Adresse der Speicherzelle aus der Summe des Operandenwerts und dem Inhalt des X-Registers gebildet. Da als Operandenwerte nur 8-Bit-Zahlen zulässig sind, haben entsprechende Befehle eine Länge von zwei Bytes.

indiziert (Y)	*addr,Y*	(8-Bit-Wert)

Im Unterschied zu der zuvor beschriebenen Adressierungsart verwendet diese das Y-Register als Index.

absolut indiziert (X)	*addr 16,X*	(16-Bit-Wert)

Auch bei dieser Adressierung wird die Adresse der Speicherzelle aus dem Operandenwert und dem Inhalt des X-Registers gebildet. Hier sind jedoch Operandenadressen mit 16 Bit erlaubt. Dementsprechend sind Befehle auch drei Bytes lang. Da das X-Register nur Werte zwischen 0 und 255 aufnimmt, kann mit dieser Adressierungsart maximal eine 256 Bytes lange Tabelle angesprochen werden.

absolut indiziert (Y)	*addr 16,Y*	(16-Bit-Wert)

Gegenüber der vorangegangenen Adressierungsart benutzt diese das Y-Register zur Bestimmung der Adresse der Speicherzelle.

nach-indiziert indirekt	*(addr),Y*	(8-Bit-Wert)

Die Bestimmung der eigentlichen Adresse erfolgt in zwei Schritten. Der Operandenwert bezeichnet eine Speicherzelle in den ersten 256 Bytes, die das LO-Byte einer Adresse enthält. Die nachfolgende Speicherzelle enthält das HI-Byte. Zu der in diesen Speicherzellen enthaltenen Adresse wird der Inhalt des Y-Registers addiert, womit die endgültige Adresse errechnet wäre. Die Adressierungsart wird in der Regel zum Ansprechen einer größeren Tabelle verwendet. Entsprechende Befehle sind zwei Bytes lang.

vor-indiziert indirekt	*(addr ,X)*	(8-Bit-Wert)

Wie bei der vorangegangenen Adressierungsart wird auch hier die Adresse der Speicherzelle in zwei Schritten bestimmt. Zu dem im Operanden angege-

benen Wert wird der Inhalt des X-Registers addiert. Dieser Wert ist die Adresse einer Zero-Page-Speicherzelle, die das LO-Byte der eigentlichen Adresse enthält. Die nachfolgende Speicherzelle enthält das HI-Byte.

indirekt *(addr 16)* (16-Bit-Wert)

Der Operandenwert zeigt auf zwei Speicherzellen, die die eigentliche Adresse enthalten, in der Reihenfolge LO-Byte und dann HI-Byte. Als einziger Befehl bietet JMP diese Adressierungsart. Bei Befehlen mit nach-indizierter indirekter Adressierung kann jedoch die indirekte Adressierung simuliert werden. Das Y-Register muß dazu nur den Wert Null enthalten.

relativ *Zieladresse*

Diese Adressierung wird nur von den Branch-Befehlen verwendet. Die eigentliche Adresse, hier die Zieladresse eines Sprungs, wird aus der aktuellen Position des Programmzählers und dem Operandenwert des Maschinenbefehls gebildet. Der Assembler erlaubt jedoch zur Vereinfachung die direkte Eingabe der Zieladresse. Der für den Prozessor benötigte relative Wert wird errechnet.

EDASS wählt bei den indizierten Adressierungen bzw. Zero-Page- und absoluter Adressierung, die sich im Eingabeformat nicht unterscheiden, die jeweils günstigere aus. Das heißt: Bei Operandenwerten kleiner als 256 wird die kürzere Zero-Page-Adressierung im Objektcode verwendet. Wird in solchen Fällen eine 16-Bit-Adresse unbedingt gewünscht, kann dies durch Voranstellen eines Ausrufezeichens erzwungen werden (z.B. LDA !121,X; STA !ERG).

Anhang C

Kurzbeschreibung des 8502-Befehlssatzes

Im folgenden Abschnitt werden alle Befehle des 8502 in alphabetischer Reihenfolge aufgeführt. Es wird jeweils eine kurze Funktionserklärung und eine Tabelle mit wichtigen Informationen zum Befehl aufgeführt. In der Spalte "Adressierung" werden die verschiedenen möglichen Adressierungsarten aufgezählt, in der Spalte "DEZ" bzw. "HEX" der Maschinencode-Wert des entsprechenden Befehlswortes.

Unter "Länge" finden Sie die Befehlslänge bei der entsprechenden Adressierungsart und schließlich unter "Taktzyklen" die Zahl der Systemtakte, die der Prozessor braucht, um den Befehl abzuarbeiten.

Die angegebene Zahl muß durch die Taktfrequenz des Prozessors dividiert werden, womit man die Ausführungszeit in Sekunden erhält. Bei fünf Taktzyklen und einer Taktfrequenz von 1 MHz, was dem SLOW-Modus entspricht, sähe diese Rechnung wie folgt aus:

5/1000000 = 0.000005

Bei einzelnen Adressierungsarten hängt die Ausführungszeit vom Operandenwert ab. Gekennzeichnet wird dies durch einen kleinen Index am Taktzyklenwert. Diese haben folgende Bedeutung:

+ Wird bei der Indizierung eine Seitengrenze, d.h. ein 256-Byte-Block, ausgewählt durch das HI-Byte, überschritten, wird vom Prozessor ein weiterer Taktzyklus benötigt.

++ Bei Branch-Befehlen muß ein Taktzyklus addiert werden, wenn der Sprung ausgeführt wird, und noch ein weiterer Taktzyklus muß addiert werden, wenn beim Sprung eine Seitengrenze, das heißt das Ende eines 256-Byte-Blocks, übersprungen wird.

Der Tabelle folgt eine Angabe, welche Flags wie verändert werden. Die Tabelleneinträge haben folgende Bedeutung:

– das Flag wird nicht verändert.
0 das Flag wird gelöscht.
1 das Flag wird gesetzt.
X das Flag enthält einen Wert entsprechend der vorangegangenen Operation

ADC – ADd with Carry to accumulator

Addiert den durch den Operanden bestimmten Wert zum Inhalt des Akkumulators. Ein Übertrag im Carry-Flag, z.B. von einer vorangegangenen Addition, wird ebenfalls addiert. Das Ergebnis steht im Akkumulator. Die Flags N,V,C und Z zeigen die entsprechenden Zustände des Ergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	105	69	2	2
addr	101	65	2	3
addr16	109	6D	3	4
addr ,X	117	75	2	4
addr16,X	125	7D	3	4+
addr16,Y	121	79	3	4+
(addr),Y	113	71	2	5+
(addr,X)	97	61	2	6

N	V	B	D	I	Z	C
X	X	–	–	–	X	X

AND – logical AND

Verknüpft den Inhalt des Akkumulators mit dem durch den Operanden bestimmten Wert mit logisch "UND". Das Ergebnis wird im Akkumulator gespeichert. Die Flags N und Z zeigen weitere Informationen über das Ergebnis an. Die Verknüpfung erfolgt nach den in folgender Tabelle abgebildeten Regeln:

A	B	A UND B
0	0	0
1	0	0
0	1	0
1	1	1

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	41	29	2	2
addr	37	25	2	3
addr16	45	2D	3	4
addr,X	53	35	2	4
addr16,X	61	3D	3	4+
addr16,Y	57	39	3	4+
(addr),Y	49	31	2	5+
(addr,X)	33	21	2	6

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

ASL – Arithmetic Shift Left

Verschiebt die Bits des Akkumulators bzw. einer Speicherzelle nach links. Das oberste Bit 7 wird in das Carry-Flag geschoben, und in das niedrigste Bit 0 wird eine Null nachgeschoben. Die Flags N und Z zeigen weitere Zustände des Ergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
A	10	0A	1	2
addr	6	06	2	5
addr16	14	0E	3	6
addr ,X	22	16	2	6
addr16 ,X	30	1E	3	7

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

BCC – Branch if Carry Clear

Verzweigt zu einer neuen Programmposition, wenn das Carry-Flag gelöscht ist. Im Maschinencode wird das Sprungziel durch die Distanz zwischen aktuellem Wert des Programmzählers und der Zieladresse bestimmt. Diese Methode wird relative Adressierung genannt. Im Assemblerprogramm wird jedoch zur Vereinfachung die Adresse des Sprungzieles direkt als Operand eingegeben.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	144	90	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BCS – Branch if Carry Set

Führt einen Sprung aus, wenn das Carry-Flag gesetzt ist. Die Zieladresse wird durch relative Adressierung bestimmt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	176	B0	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BEQ – Branch if EQual to zero

Führt einen Sprung durch, wenn das Zero-Flag gesetzt ist. Das Sprungziel wird durch relative Adressierung bestimmt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	240	F0	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BIT – BIT-test with accumulator

Prüft durch eine logische "UND"-Verknüpfung mit dem durch den Operanden bestimmten Wert die Bits des Akkumulators. Das N- bzw. V-Flag zeigt den Wert von Bit 7 bzw. Bit 6 des Operandenwertes an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr	36	24	2	3
addr16	44	2C	3	4

N	V	B	D	I	Z	C
X	X	–	–	–	X	–

BMI – Branch if MInus

Verzweigt zu einer neuen Programmposition, wenn das Negativ-Flag gesetzt ist. Im Maschinencode wird als Operand die Sprungdistanz angegeben, im Assemblerprogramm jedoch direkt die Zieladressse des Sprungs.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	48	30	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BNE – Branch if Not Equal to zero

Führt einen Sprung zu einer neuen Programmposition aus, wenn das Zero-Flag gelöscht ist. Durch relative Adressierung wird die neue Adresse des Programmzählers mitgeteilt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	208	D0	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BPL – Branch if PLus

Verzweigt, wenn das Negativ-Flag gelöscht ist. Die neue Adresse des Programmzählers wird durch relative Adressierung bestimmt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	16	10	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BRK – BReaK

Löst softwaremäßig einen Interrupt aus. Der Prozessor liest aus den Speicherzellen $FFFE und $FFFF die Startadresse der Interrupt-Routine und springt zu dieser. Auf dem Stapel wird die Adresse des nächsten Befehls und das Statusregister abgelegt. Um den softwaremäßigen von einen hardwaremäßigen Interrupt zu unterscheiden, wird das Break-Flag gesetzt. Um nachfolgende Interrupts zu sperren, wird auch das Interrupt-Flag gesetzt. Im Normalbetrieb ruft der BRK-Befehl den Monitor des C128 auf.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	0	00	1	7

N	V	B	D	I	Z	C
–	–	1	–	1	–	–

BVC – Branch if oVerflow Clear

Verzweigt zu einer neuen Programmadresse, wenn das Overflow-Flag gelöscht ist. Mit relativer Adressierung wird die Zieladresse des Sprungs bestimmt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	80	50	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

BVS – Branch if oVerflow Set

Führt einen Sprung aus, wenn das Overflow-Flag gesetzt ist. Das Ziel des Sprungs wird durch relative Adressierung ausgewählt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
relativ	112	70	2	2++

N	V	B	D	I	Z	C
keine Veränderung						

CLC – CLear Carry

Löscht das Carry-Flag. Bei einer Addition ist es unerwünscht, daß ein alter Inhalt des Carry-Flags mit addiert wird. Vor einer Addition wird deshalb häufig mit diesem Befehl ein Übertrag gelöscht.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	24	18	1	2

N	V	B	D	I	Z	C
–	–	–	–	–	–	0

CLD – CLear Decimal mode

Löscht das Dezimal-Flag. Ein zuvor eingeschalteter Dezimal-Modus wird mit diesem Befehl abgeschaltet.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	216	D8	1	2

N	V	B	D	I	Z	C
–	–	–	0	–	–	–

CLI – CLear Interrupt enable

Löscht das Interrupt-Flag. Die Unterbrechung durch einen Interrupt wird mit diesem Befehl wieder erlaubt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	88	58	1	2

N	V	B	D	I	Z	C
–	–	–	–	0	–	–

CLV – CLear oVerflow

Löscht das Overflow-Flag. Benötigt wird dieser Befehl im praktischen Einsatz fast nie.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	184	B8	1	2

N	V	B	D	I	Z	C
–	0	–	–	–	–	–

CMP – CoMPare memory with accumulator

Vergleicht den Inhalt des Akkumulators mit dem durch den Operanden bestimmten Wert. Der Vergleich entspricht einer Subtraktion. Das Ergebnis wird jedoch nur in den Statusflags angezeigt. Folgende verschiedene Zustände sind möglich:

	Z	C
Akkumulator < Wert	0	0
Akkumulator = Wert	1	1
Akkumulator > Wert	0	1

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	201	C9	2	2
addr	197	C5	2	3
addr16	205	CD	3	4
addr ,X	213	D5	2	4
addr16 ,X	221	DD	3	4+
addr16 ,Y	217	D9	3	4+
(addr),Y	209	D1	2	5+
(addr ,X)	193	C1	2	6

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

CPX – ComPare X-register with memory

Vergleicht den Inhalt des X-Registers mit einem Wert im Speicher. Zum Vergleich wird eine Subtraktion durchgeführt. Deren Ergebnis wird nur in den Statusflags angezeigt. Die verschiedenen Ergebnisse sehen Sie bei der Erklärung des CMP-Befehls.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	224	E0	2	2
addr	228	E4	2	3
addr16	236	EC	3	4

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

CPY – ComPare Y-register with memory

Vergleicht den Inhalt des Y-Registers mit dem Inhalt einer Speicherzelle, deren Adresse durch den Operanden bestimmt wird. Der Vergleich entspricht einer Subtraktion, bei der als Ergebnis nur die Flags gesetzt bzw. gelöscht werden.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	192	C0	2	2
addr	196	C4	2	3
addr16	204	CC	3	4

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

DEC – DECrement memory

Dekrementiert vermindert den Inhalt einer Speicherzelle um eins. Das Z- und das N-Flag zeigen den Zustand des Ergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr	198	C6	2	5
addr16	206	CE	3	6
addr ,X	214	D6	2	6
addr16 ,X	222	DE	3	7

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

DEX – DEcrement X-register

Vermindert den Inhalt des X-Registers um eins. Die Flags zeigen entsprechend den Zustand des Endergebnisses der Operation an, d.h. das Z-Flag ist gesetzt, wenn das Ergebnis null ist. Das N-Flag ist gesetzt, wenn das Ergebnis negativ ist, also im Zahlenbereich von 128 bis 255 liegt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	202	CA	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

DEY – DEcrement Y-register

Vermindert den Inhalt des Y-Registers um eins. Die Statusflags Z und N zeigen den Zustand des Endergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	136	88	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

EOR – Exclusiv-OR with accumulator and memory

Verknüpft den Inhalt des Akkumulators mit dem Inhalt der Speicherzelle, die durch den Operanden adressiert wird, logisch "exklusiv oder". Die Flags Z und N zeigen an, ob das Ergebnis identisch mit null bzw. negativ ist. Die Verknüpfung der einzelnen Bits erfolgt nach folgender Wertetabelle:

A	B	A EOR B
0	0	0
1	0	1
0	1	1
1	1	0

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	73	49	2	2
addr	69	45	2	3
addr16	77	4D	3	4
addr ,X	85	55	2	4
addr16 ,X	93	5D	3	4+
addr16 ,Y	89	59	3	4+
(addr),Y	81	51	2	5+
(addr,X)	65	41	2	6

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

INC – INCrement memory

Inkrementiert (erhöht um eins) den Inhalt einer Speicherzelle. Die Flags N und Z zeigen entsprechend den Zustand des Ergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr	230	E6	2	5
addr16	238	EE	3	6
addr ,X	246	F6	2	6
addr16 ,X	254	FE	3	7

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

INX – INcrement X-register

Erhöht den Inhalt des X-Registers um eins. Die Flags Z und N zeigen an, ob das Ergebnis der Operation identisch mit null ist bzw. einen negativen Wert besitzt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	232	E8	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

INY – INcrement Y-register

Erhöht den Inhalt des X-Registers um eins. Die Status-Flags N und Z zeigen bestimmte Zustände des Endergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	200	C8	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

JMP – JuMP

Springt zu einer neuen Programmposition. Der Sprungbefehl ist mit einem Laden des Programmzählers zu vergleichen. Dieser bestimmt, von welcher Adresse das nächste Befehlsbyte gelesen wird.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr16	76	4C	3	3
(addr16)	108	6C	3	5

N	V	B	D	I	Z	C
keine Veränderung						

JSR – Jump to SubRoutine

Ruft ein Unterprogramm auf. Der eigentliche Sprung wird wie beim JMP-Befehl ausgeführt, zuvor wird jedoch der aktuelle Wert des Programmzählers auf den Stapel gerettet. Der aktuelle Wert entspricht in diesem Fall der Adresse des dritten Bytes im JSR-Befehl bzw. umgerechnet der Adresse des nächsten Befehls minus eins. Ein Unterprogramm wird am Ende wieder mit RTS verlassen.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr16	32	20	3	6

N	V	B	D	I	Z	C
keine Veränderung						

LDA – LoaD Accumulator

Lädt einen Wert in den Akkumulator. Dieser Ladevorgang entspricht einem Kopieren, denn der Inhalt der Speicherzelle, aus der die Daten stammen, bleibt erhalten. Beim Laden werden die Status-Flags Z und N entsprechend dem geladenen Wert gesteuert.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	169	A9	2	2
addr	165	A5	2	3
addr16	173	AD	3	4
addr ,X	181	B5	2	4
addr16 ,X	189	BD	3	4+
addr16 ,Y	185	B9	3	4+
(addr),Y	177	B1	2	5+
(addr ,X)	161	A1	2	6

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

LDX – LoaD X-register

Überträgt den Inhalt einer Speicherzelle in das X-Register. Die Flags N und Z geben Auskünfte über den Wert selbst.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	162	A2	2	2
addr	166	A6	2	3
addr16	174	AE	3	4
addr ,Y	182	B6	2	4
addr16 ,Y	190	BE	3	4+

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

LDY – LoaD Y-register

Lädt das Y-Register mit dem Inhalt einer Speicherzelle. Aussagen über den geladenen Wert lassen die Zustände der Flags N und Z zu.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	160	A0	2	2
addr	164	A4	2	3
addr16	172	AC	3	4
addr ,X	180	B4	2	4
addr16 ,X	188	BC	3	4+

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

LSR – Logical Shift Right

Verschiebt die Bits des Akkumulators bzw. einer Speicherzelle nach rechts. Bit 0 wird ins Carry-Flag geschoben, und in Bit 7 wird eine Null nachgeschoben. Neben dem Carry-Flag zeigen auch die Flags Z und N einen Zustand des Endergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
A	74	4A	1	2
addr	70	46	2	5
addr16	78	4E	3	6
addr ,X	86	56	2	6
addr16 ,X	94	5E	3	7

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

NOP – No OPeration

Veranlaßt den Prozessor für die Dauer des Befehls, nämlich zwei Taktzyklen, nichts zu tun. In der Assemblerprogrammierung hat dieser Befehl wenig Bedeutung.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	234	EA	1	2

N	V	B	D	I	Z	C
keine Veränderung						

ORA – logical OR with Accumulator and memory

Verknüpft den Inhalt des Akkumulators mit dem Inhalt einer Speicherzelle logische "ODER". Die Flags N und Z geben Auskunft über den Wert des Ergebnisses. Die Verknüpfung folgt den Regeln folgender Tabelle:

A	B	A ODER B
0	0	0
1	0	1
0	1	1
1	1	1

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	9	09	2	2
addr	5	05	2	3
addr16	13	0D	3	4
addr ,X	21	15	2	4
addr16 ,X	29	1D	3	4+
addr16 ,Y	25	19	3	4+
(addr),Y	17	11	2	5+
(addr ,X)	1	01	2	6

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

PHA – PusH Accummulator onto stack

Speichert den Inhalt des Akkumulators an die oberste Position des Stapels und setzt den Stapelzeiger auf die neue Ablageposition. Der Befehl wird hauptsächlich verwendet, um den Inhalt des Akkumulators zu sichern und ihn an einer anderen Programmposition wieder zurückzuholen.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	72	48	1	3

N	V	B	D	I	Z	C
keine Veränderung						

PHP – PusH statusregister (P) onto stack

Speichert den Inhalt des Statusregisters auf die oberste Position des Stapels. Dies ist der einzige Weg, alle acht Bits des Statusregisters auf einmal auszulesen.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	8	08	1	3

N	V	B	D	I	Z	C
keine Veränderung						

PLA – PuLl Accumulator from stack

Liest den obersten Wert des Stacks und speichert diesen im Akkumulator. Die Status-Flags N und Z werden entsprechend dem gelesenen Wert gesetzt. Dieser Befehl ist das Gegenstück zum PHA-Befehl und dient hauptsächlich zum Zurückholen eines zwischengespeicherten Akkumulatorinhalts.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	104	68	1	4

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

PLP – PuLl statusregister (P) from stack

Liest den obersten Wert vom Stapel und speichert diesen in das Statusregister. Dieser Befehl ist die einzige Möglichkeit, alle acht Bits des Statusregisters auf einmal zu beschreiben. Es wird deshalb auch der Inhalt aller Flags beeinflußt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	40	28	1	4

N	V	B	D	I	Z	C
X	X	X	X	X	X	X

ROL – ROtate Left

Läßt die Bits des Akkumulators bzw. einer Speicherzelle nach links rotieren. In Bit 0 wird der Inhalt des Carry-Flags geschoben, und der Inhalt von Bit 7 wird in das Carry-Flag geschoben. Nach neunmaliger Anwendung des ROL-Befehls auf den Akkumulator bzw. eine Speicherzelle ist der ursprüngliche Inhalt wieder hergestellt. Neben dem Carry-Flag zeigen die Flags N und Z Zustände des Ergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
A	42	2A	1	2
addr	38	26	2	5
addr16	46	2E	3	6
addr ,X	54	36	2	6
addr16 ,X	62	3E	3	7

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

ROR – ROtate Right

Rotiert die Bits des Akkumulators oder einer Speicherzelle über das Carry-Flag nach rechts. In Bit 7 wird der Inhalt des Carry-Flags nachgeschoben, und Bit 0 wird ins Carry-Flag geschoben. Nach neun Rotationen ist der ursprüngliche Inhalt des Akkumulators bzw. der Speicherzelle wiederhergestellt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
A	106	6A	1	2
addr	102	66	2	5
addr16	110	6E	3	6
addr ,X	118	76	2	6
addr16 ,X	126	7E	3	7

N	V	B	D	I	Z	C
X	–	–	–	–	X	X

RTI – ReTurn from Interrupt

Dieser Befehl dient zum Rücksprung aus einer Interrupt-Routine in das unterbrochene Programm. Vom Stapel wird der ehemalige Inhalt des Programm-

zählers und des Statusregisters gelesen. Aus diesem Grund werden auch die Inhalte aller Flags verändert.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	64	40	1	6

N	V	B	D	I	Z	C
X	X	X	X	X	X	X

RTS – ReTurn from Subroutine

Dient zum Verlassen eines mit JSR aufgerufenen Unterprogramms. Vom Stapel wird die Adresse gelesen, an der das Hauptprogramm verlassen wurde.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	96	60	1	6

N	V	B	D	I	Z	C
keine Veränderung						

SBC – SuBtract from accumulator with Carry

Subtrahiert den Inhalt einer Speicherzelle vom Inhalt des Akkumulators und speichert in diesen das Ergebnis. Bei der Subtraktion wird ein Übertrag, z.B. von einer vorangegangenen Subtraktion, verrechnet. Zu beachten ist, daß das Übertrags-Flag gesetzt ist, wenn kein Übertrag auftrat. Deshalb muß auch mit SEC das Carry-Flag gesetzt werden, wenn in eine Rechnung kein Übertrag ein- fließen soll. Die Flags Z, N, V und C zeigen verschiedene Zustände des Ergebnisses an.

Adressierung	DEZ	HEX	Länge	Taktzyklen
# data	233	E9	2	2
addr	229	E5	2	3
addr16	237	ED	3	4
addr ,X	245	F5	2	4
addr16 ,X	253	FD	3	4+
addr16 ,Y	249	F9	3	4+
(addr),Y	241	F1	2	5+
(addr ,X)	225	E1	2	6

N	V	B	D	I	Z	C
X	X	–	–	–	X	X

SEC – SEt Carry

Setzt das Carry-Flag. Der Befehl ist häufig vor einer Subtraktion notwendig, um zu verhindern, daß ein Übertrag in die Rechnung einfließt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	56	38	1	2

N	V	B	D	I	Z	C
–	–	–	–	–	–	1

SED – SEt Decimal mode

Setzt das Dezimal-Flag. Der Prozessor wird damit auf Dezimalarithmetik umgeschaltet. Die Befehle ADC und SBC verarbeiten dann BCD-Zahlen.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	248	F8	1	2

N	V	B	D	I	Z	C
–	–	–	1	–	–	–

SEI – SEt Interrupt enable

Setzt das Interrupt-Flag. Es wird damit eine Unterbrechung durch Interrupts unterbunden.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	120	78	1	2

N	V	B	D	I	Z	C
–	–	–	–	1	–	–

STA – STore Accumulator

Speichert den Inhalt des Akkumulators in eine Speicherzelle, deren Adresse durch den Operanden bestimmt wird. Die Inhalte der Statusflags werden dabei nicht verändert.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr	133	85	2	3
addr16	141	8D	3	4
addr,X	149	95	2	4
addr16,X	157	9D	3	5
addr16,Y	153	99	3	5
(addr),Y	145	91	2	6
(addr,X)	129	81	2	6

N	V	B	D	I	Z	C
keine Veränderung						

STX – STore X-register

Speichert den Inhalt des X-Registers in eine Speicherzelle. Die Inhalte der Status-Bits bleiben unberührt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr	134	86	2	3
addr16	142	8E	3	4
addr,Y	150	96	2	4

N	V	B	D	I	Z	C
keine Veränderung						

STY – STore Y-register

Speichert den Inhalt des Y-Registers in eine Speicherzelle. Die Status-Flags behalten dabei ihren Wert.

Adressierung	DEZ	HEX	Länge	Taktzyklen
addr	132	84	2	3
addr16	140	8C	3	4
addr,X	148	94	2	4

N	V	B	D	I	Z	C
keine Veränderung						

TAX – Transfer from Accumulator to X-register

Kopiert den Inhalt des Akkumulators in das X-Register. Es ist zu beachten, daß der Inhalt kopiert und nicht vertauscht wird. Die Flags N und Z werden entsprechend dem übertragenen Wert gesetzt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	170	AA	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

TAY – Transfer from Accumulator to Y-register

Kopiert den Inhalt des Akkumulators in das Y-Register. Die Flags N und Z werden entsprechend dem übertragenen Wert gesetzt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	168	A8	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

TSX – Transfer Stack-pointer to X-register

Überträgt den Inhalt des Stapelzeigers, der auf die aktuelle Ablageposition im Stapel zeigt, in das X-Register. Dies ist der einzige Befehl, die aktuelle Stapelposition festzustellen. Die Flags N und Z werden entsprechend dem übertragenen Wert gesetzt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	186	BA	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

TXA – Transfer from X-register to Accumulator

Kopiert den Inhalt des X-Registers in den Akkumulator. Entsprechend dem übertragenen Wert werden die Flags N und Z gesetzt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	138	8A	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

TXS – Transfer from X-Register to Stack-pointer

Überträgt den Inhalt des X-Registers in den Stapelzeiger. Dies ist der einzige Befehl, mit dem die Position des Stapels verändert werden kann. Die Flags N und Z werden entsprechend dem übertragenen Wert beeinflußt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	154	9A	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

TYA – Transfer from Y-register to Accumulator

Kopiert den Inhalt des Y-Registers in den Akkumulator. Die Status-Flags N und Z werden entsprechend dem übertragenen Wert gesetzt.

Adressierung	DEZ	HEX	Länge	Taktzyklen
implizit	152	98	1	2

N	V	B	D	I	Z	C
X	–	–	–	–	X	–

Anhang D

8502-Befehlssatz in numerischer Reihenfolge

Objektcode dez.	hex.	Mnemonik	Adressierart
0	00	brk	
1	01	ora	(addr,x)
5	05	ora	addr
6	06	asl	addr
8	08	php	
9	09	ora	data
10	0a	asl	a
13	0d	ora	addr 16
14	0e	asl	addr 16
16	10	bpl	disp
17	11	ora	(addr),y
21	15	ora	addr,x
22	16	asl	addr,x
24	18	clc	
25	19	ora	addr 16,y
29	1d	ora	addr 16,x
30	1e	asl	addr 16,x
32	20	jsr	addr 16
33	21	and	(addr,x)
36	24	bit	addr
37	25	and	addr
38	26	rol	addr
40	28	plp	
41	29	and	data
42	2a	rol	a
44	2c	bit	addr 16
45	2d	and	addr 16
46	2e	rol	addr 16
48	30	bmi	disp
49	31	and	(addr),y
53	35	and	addr,x

Objektcode dez.	hex.	Mnemonik	Adressierart
54	36	rol	addr,x
56	38	sec	
57	39	and	addr 16,y
61	3d	and	addr 16,x
62	3e	rol	addr 16,x
64	40	rti	
65	41	eor	(addr,x)
69	45	eor	addr
70	46	lsr	addr
72	48	pha	
73	49	eor	data
74	4a	lsr	a
76	4c	jmp	addr 16
77	4d	eor	addr 16
78	4e	lsr	addr 16
80	50	bvc	disp
81	51	eor	(addr),y
85	55	eor	addr,x
86	56	lsr	addr,x
88	58	cli	
89	59	eor	addr 16,y
93	5d	eor	addr 16,x
94	5e	lsr	addr 16,x
96	60	rts	
97	61	adc	(addr,x)
101	65	adc	addr
102	66	ror	addr
104	68	pla	
105	69	adc	data
106	6a	ror	a
108	6c	jmp	(addr 16)
109	6d	adc	addr 16
110	6e	ror	addr 16
112	70	bvs	disp
113	71	adc	(addr),y
117	75	adc	addr,x
118	76	ror	addr,x
120	78	sei	
121	79	adc	addr 16,y
125	7d	adc	addr 16,x

Objektcode dez.	hex.	Mnemonik	Adressierart
126	7e	ror	addr 16,x
129	81	sta	(addr,x)
132	84	sty	addr
133	85	sta	addr
134	86	stx	addr
136	88	dey	
138	8a	txa	
140	8c	sty	addr 16
141	8d	sta	addr 16
142	8e	stx	addr 16
144	90	bcc	disp
145	91	sta	(addr),y
148	94	sty	addr,x
149	95	sta	addr,x
150	96	stx	add,y
152	98	tya	
153	99	sta	addr 16,y
154	9a	txs	
157	9d	sta	addr 16,x
160	a0	ldy	data
161	a1	lda	(addr,x)
162	a2	ldx	data
164	a4	ldy	addr
165	a5	lda	addr
166	a6	ldx	addr
168	a8	tay	
169	a9	lda	data
170	aa	tax	
172	ac	ldy	addr 16
173	ad	lda	addr 16
174	ae	ldx	addr 16
176	b0	bcs	disp
177	b1	lda	(addr),y
180	b4	ldy	addr,x
181	b5	lda	addr,x
182	b6	ldx	addr,y
184	b8	clv	
185	b9	lda	addr 16,y
186	ba	tsx	
188	bc	ldy	addr 16,x

Objektcode dez.	hex.	Mnemonik	Adressierart
189	bd	lda	addr 16,x
190	be	ldx	addr 16,y
192	c0	cpy	data
193	c1	cmp	(addr,x)
196	c4	cpy	addr
197	c5	cmp	addr
198	c6	dec	addr
200	c8	iny	
201	c9	cmp	data
202	ca	dex	
204	cc	cpy	addr 16
205	cd	cmp	addr 16
206	ce	dec	addr 16
208	d0	bne	disp
209	d1	cmp	(addr),y
213	d5	cmp	addr,x
214	d6	dec	addr,x
216	d8	cld	
217	d9	cmp	addr 16,y
221	dd	cmp	addr 16,x
222	de	dec	addr 16,x
224	e0	cpx	data
225	e1	sbc	(addr,x)
228	e4	cpx	addr
229	e5	sbc	addr
230	e6	inc	addr
232	e8	inx	
233	e9	sbc	data
234	ea	nop	
236	ec	cpx	addr 16
237	ed	sbc	addr 16
238	ee	inc	addr 16
240	f0	beq	disp
241	f1	sbc	(addr),y
245	f5	sbc	addr,x
246	f6	inc	addr,x
248	f8	sed	
249	f9	sbc	addr 16,y
253	fd	sbc	addr 16,x
254	fe	inc	addr 16,x

Anhang E

EDASS-Befehle

Bei der Verwendung der Befehle muß bei der Eingabe immer ein Ausrufezeichen vorangestellt werden, um mitzuteilen, daß ein EDASS-Befehl eingegeben wurde. Mit dem MOD-Befehl kann dieses Einleitungszeichen geändert werden.

Befehle	Abkürzungen
=	=

wertet einen nachfolgenden math. Ausdruck aus und stellt das Ergebnis dezimal dar.

$=	**$=**

wertet einen nachfolgenden math. Ausdruck aus und stellt das Ergebnis hexadezimal dar.

%=	**%=**

wertet einen nachfolgenden math. Ausdruck aus und stellt das Ergebnis binär dar.

= / @=	**= / @=**

wertet einen nachfolgenden math. Ausdruck aus und stellt das Ergebnis oktal dar.

ASSEMBLER	**aS**

ruft den Assembler auf, um ein Programm zu assemblieren.

BEGIN	**bE**

teilt dem Editor mit, daß ein neues Programm im Speicher angefangen wird.

BYTE **bY**

wandelt einen Speicherbereich in eine edierfähige Tabelle aus Pseudoanweisungen .BYTE um.

CLEAR **cL**

löscht alle Labels im Speicher.

COLD **cO**

deaktiviert EDASS. Der gesamte Speicher wird BASIC wieder zugänglich gemacht. Ein BASIC-Programm wird nicht gelöscht.

DISPLAY **dI**

gibt eine Namensliste aller im Speicher befindlicher Programme aus.

EDIT **eD**

ruft den Editor auf, um das angegebene Programm zu edieren.

ERASE **eR**

löscht ein komplettes Programm bzw. im Editor einen Teil eines Programms.

EXIT **eX**

verläßt den Editor wieder.

FIND **fI**

sucht eine Zeichenfolge im Programm. Auf Wunsch kann diese durch eine andere ersetzt werden.

GO **go**

startet ein Maschinenprogramm im Computerspeicher.

GOD **god**

startet ein Maschinenprogramm im Speicher des Diskettenlaufwerks.

IMPORT **iM**

liest ein Programm, das nicht im EDASS-Format gespeichert ist, ein.

INSERT **iN**

fügt einen Teil bzw. ein ganzes Programm in den aktuellen Programmtext des Editors ein.

JUMP **jU**

springt zur angegebenen Zeilennummer des Programms.

LET **lE**

definiert im Direktmodus ein Label und weist diesem einen Wert zu.

LIST **lI**

gibt den Text eines angegebenen Programms aus.

LOAD **lO**

lädt ein Programm in den Arbeitsspeicher von EDASS.

LBL$ **lB**

gibt im Direktmodus eine Liste aller im Speicher befindlicher Labels aus. Deren Werte werden hexadezimal dargestellt.

LBL **lbl**

gibt im Direktmodus eine Liste aller im Speicher befindlicher Labelwerte aus. Diese werden dezimal dargestellt.

KILL **kI**

löscht im Direktmodus ein einzelnes im Speicher befindliches Label.

MOD **mO**

verändert mehrere Parameter, die die Arbeitsweise von EDASS beeinflussen.

NEW **nE**

teilt den gesamten verfügbaren Speicher neu zwischen EDASS und BASIC auf.

PUSH **pU**

speichert im Direktmodus alle im Speicher befindlichen Labels auf Diskette.

PULL **puL**

liest im Direktmodus eine zuvor mit PUSH bzw. .PUSH gespeicherte Labeltabelle wieder ein.

REASS **reA**

startet den Reassembler, um ein edierfähiges Programm zu erzeugen.

RENAME **rE**

benennt ein im Speicher befindliches Programm um.

SAVE **sA**

speichert ein Assemblerprogramm auf Dis
kette ab.

STORE **sT**

speichert die mit MOD und NEW veränderbaren Parameter und die aktuelle Funktionstastenbelegung.

Anhang F

Pseudo-Befehle

Pseudo-Befehl	Abkürzung
=	=
definiert ein neues Label und weist diesem einen Wert zu.	
< /	< /
gibt einem bereits definierten Label einen neuen Wert.	
/	/
ruft ein zuvor definiertes Makro auf.	
*=	*=
setzt den Programmzeiger des Assemblers auf einen neuen Wert.	
.BYTE	**.bY**
fügt einzelne 8-Bit-Werte in den Objektcode ein.	
.BANK	**.bA**
bestimmt, in welche Speicherbank der Objektcode geschrieben werden soll.	
.DBYTE	**.dB**
baut einzelne 16-Bit-Werte in der Reihenfolge HI-Byte, LO-Byte, in den Objektcode ein.	
.DISK	**.dI**
sendet einen Befehl an eine Diskettenstation.	

.END **.eN**
markiert das Ende eines Quelltextes.

.ENDM **.endm**

beendet die Definition und den Aufruf eines Makros.

.ERROR **.eR**

leitet alle Fehlermeldungen auf ein bestimmtes Gerät und unterbindet den Abbruch der Assemblierung bei einem Fehler.

.FAST .fA

schaltet während der Assemblierung den C128 auf die schnellere Betriebsart.

.FF **.ff**

erzwingt bei formatierter Ausgabe des Listings einen Seitenvorschub.

.FILE **.fI**

verknüpft zwei Programme auf eine besondere Weise.

.FORMAT **.fO**

veranlaßt den Assembler ein Listing formatiert auszugeben.

.IF / .ENDIF **.if / .endI**

assembliert die eingeschlossenen Programmzeilen nur, wenn ein nachfolgender mathematischer Ausdruck ungleich Null ist.

.IFEQ **.ifE**

leitet eine bedingte Assemblierung mit der Bedingung "gleich Null" ein.

.IFMI **.ifM**

leitet eine bedingte Assemblierung mit der Bedingung "negativ" ein.

.IFNE **.ifN**

leitet eine bedingte Assemblierung mit der Bedingung "ungleich Null" ein.

.IFPL **.ifP**

leitet eine bedingte Assemblierung mit der Bedingung "positiv" ein.

LBL$ **.lB**

gibt eine Liste aller im Speicher befindlicher Labels aus. Deren Werte erscheinen hexadezimal.

.LBL **.lbl**

gibt eine Liste aller im Speicher befindlicher Labels aus. Die Labelwerte erscheinen dezimal.

.LIST **.lI**

veranlaßt den Assembler ein Assemblerlisting auszugeben.

.LISTM **.listm**

gibt im Assemblerlisting die assemblierten Makros aus.

.MACRO **.mA**

leitet die Definition eines Makros ein.

.OBJ **.oB**

veranlaßt den Assembler den Objektcode in den Speicher oder auf ein Peripheriegerät zu schreiben.

.OFF **.oF**

bestimmt eine Differenz zwischen dem Programmzeiger des Assemblers und der physikalischen Schreibadresse des Objektcodes.

.PAGE **.pA**

startet bei formatierter Ausgabe des Assemblerlistings die Ausgabe einer Seitennummer.

.PRINT$ **.pR**

gibt im Assemblerlisting Texte und einzelne Labelwerte aus. Diese werden hexadezimal dargestellt.

.PRINT **.print**

gibt im Assemblerlisting Texte und einzelne Labelwerte aus. Diese werden dezimal dargestellt.

.PULL .puL

liest eine Labeltabelle von Diskette ein.

.PUSH **.pU**

schreibt eine Labeltabelle auf Diskette

.SLOW **.sL**

schaltet während des Assemblierens auf die langsamere Betriebsart des C128.

.TEXT **.tE**

baut Zeichenfolgen und 8-Bit-Werte in den Objektcode ein.

.TITLE **.tI**

veranlaßt den Assembler bei formatierter Ausgabe des Listings, eine Überschrift auf jede Seite zu schreiben.

.VIDEO **.vI**

baut Zeichenfolgen und 8-Bit-Werte, umgewandelt in den CBM-Bildschirmcode, in den Objektcode ein.

.WORD **.wO**

fügt 16-Bit-Werte in der Reihenfolge LO-Byte, HI-Byte in den Objektcode ein.

.WRITE **.wR**

gibt Texte und Steuerzeichen im Assemblerlisting aus.

Anhang G

Freie Speicherzellen in der Zero-Page

Die Speicherzellen 0–255 in der sogenannten Zero-Page haben eine besondere Bedeutung bei Adressierungsarten. Ihre Verwendung im Operand verkürzt die Befehlslänge oder ermöglicht z.B. erst die "nach-indizierte-indirekte" Adressierung. Wegen dieser Vorteile sind die 256 Speicherzellen in der Zero-Page in Programmen sehr begehrt.

Der BASIC-Interpreter und das Betriebssystem belegen nahezu alle 256 Speicherplätze. Trotzdem stehen Ihnen noch zahlreiche Speicherzellen, wenn auch mit Einschränkungen, für eigene Programme zur Verfügung.

Es folgt eine Aufstellung dieser Speicherzellen, geordnet nach den diversen Kategorien der Einschränkungen: Die folgenden Speicherzellen sind ohne Einschränkungen benutzbar. Sie können zur Parameterübergabe sowohl zwischen BASIC- und Maschinenprogramm als auch zur Zwischenspeicherung von Werten genutzt werden.

Speicherzellen 250–255 ($FA–$FF)

Sehr viele Speicherzellen, die vom BASIC-Interpreter belegt sind, werden nur für spezielle Aufgaben benutzt und liegen deshalb den größten Teil der Zeit brach. Diese Speicherzellen können zur Parameterübergabe und zum Zwischenspeichern von Werten genutzt werden. Es sollte beachtet werden, daß zur Übergabe von Werten diese direkt vor dem Aufruf des Maschinenprogramms in die Speicherzellen geschrieben werden und direkt nach Abschluß des Maschinenprogramms wieder ausgelesen werden.

Speicherzellen 9–13 ($09–$0D), 17–20 ($11–$14), 63 ($3F), 75–76 ($4B–$4C), 82 ($52), 85 ($55), 119–120 ($77–$78)

Unter den Speicherzellen, die vom BASIC-Interpreter genutzt werden, gibt es eine zweite Kategorie. Zu dieser gehören alle Speicherplaetze, in denen der BASIC-Interpreter während seiner Arbeit ständig Zwischenwerte ablegt. Ent-

sprechend können diese Adressen nicht zur Parameterübergabe genutzt werden, sondern stehen nur dem Maschinenprogramm zur Ablage von Werten zur Verfügung.

Speicherzellen **14–16 ($0E–$10), 21–23 ($15–$17), 36–43 ($24–$2B), 71–74 ($47–$4A), 77–84 ($4D–$54), 87–112 ($57–$70)**

Einen großen Teil der Speicherzellen in der Zero-Page werden vom Betriebssytem belegt. Solange kein BASIC-Kommando wie LOAD, SAVE oder OPEN beziehungsweise im Maschinenprogramm nicht die entsprechenden Routinen des Betriebssytems aufgerufen werden, kann über folgende Speicherzellen frei verfügt werden:

Speicherzellen **163–177 ($A3–$B1), 189–196 ($BD–$C4)**

Anhang H

Nützliche Tabellen

40-Zeichen-Modus Farbcode

0	Schwarz	8	Orange
1	Weiß	9	Braun
2	Rot	10	Hellrot
3	Cyan	11	Dunkelgrau
4	Violett	12	Mittelgrau
5	Grün	13	Hellgrün
6	Blau	14	Hellblau
7	Gelb	15	Hellgrau

Die Farbcodes belegen die unteren vier Bits eines Bytes. Beim Schreiben eines Codes ist der Wert der oberen vier Bits ohne Bedeutung.

80-Zeichen-Modus Farbcodes (Attribut-Byte)

0	Schwarz	8	Rot
1	Grau	9	Rosa
2	Blau	10	Hellbraun
3	Hellblau	11	Violett
4	Dunkelgrün	12	Braun
5	Hellgrün	13	Gelb
6	Dunkelgrau	14	Hellgrau
7	Grün	15	Weiß

Die unteren vier Bits enthalten die Farbinformation. Den oberen vier Bits kommt folgende Bedeutung zu.

Bit 4 – Blinken des Zeichens ein (1) bzw. ausschalten (0)
Bit 5 – Unterstreichen des Zeichens ein (1) bzw. ausschalten(0)
Bit 6 – Darstellung des Zeichens normal (0) bzw. invers (1)
Bit 7 – Zeichensatz Grafik/Groß (0) bzw. klein/groß wählen

Commodore Bildschirmcodes

Satz 1	Satz 2	Poke	Satz 1	Satz 2	Poke	Satz 1	Satz 2	Poke
@		0	V	v	22	,		44
A	a	1	W	w	23	–		45
B	b	2	X	x	24	.		46
C	c	3	Y	y	25	/		47
D	d	4	Z	z	26	0		48
E	e	5	[		27	1		49
F	f	6	£		28	2		50
G	g	7	]		29	3		51
H	h	8	↑		30	4		52
I	i	9	←		31	5		53
J	j	10	SPACE		32	6		54
K	k	11	!		33	7		55
L	l	12	"		34	8		56
M	m	13	#		35	9		57
N	n	14	$		36	:		58
O	o	15	%		37	;		59
P	p	16	&		38	<		60
Q	q	17	'		39	=		61
R	r	18	(		40	>		62
S	s	19	)		41	?		63
T	t	20	*		42			
U	u	21	+		43			

Commodore ASCII-Code

Satz 1	Satz 2	Poke	Satz 1	Satz 2	Poke	Satz 1	Satz 2	Poke
		64		V	86			108
	A	65		W	87			109
	B	66		X	88			110
	C	67		Y	89			111
	D	68		Z	90			112
	E	69			91			113
	F	70			92			114
	G	71			93			115
	H	72			94			116
	I	73			95			117
	J	74	SPACE		96			118
	K	75			97			119
	L	76			98			120
	M	77			99			121
	N	78			100		✓	122
	O	79			101			123
	P	80			102			124
	Q	81			103			125
	R	82			104			126
	S	83			105			127
	T	84			106			
	U	85			107			

Die Codes 128-255 ergeben die invers dargestellten Zeichen der Codes 0-127.

An den freien Stellen in Spalte 2 stimmen die beiden Sätze überein.

Commodore 128 Codetabelle

Dez	Hex	Funktion
0		
1		
2	02	Unterstreichen ein (80 Zeichen-Bildschirm)
3		
4		
5	05	Farbe weiß
6		
7	07	Klingelzeichen
8		
9	09	Tabulator
10	0a	Zeilenvorschub
11	0b	blockiert Umschaltung Text/ Grafik-Zeichensatz
12	0c	entriegelt Umschaltung Text/ Grafik-Zeichensatz
13	0d	RETURN (Wagenrücklauf und Zeilenvorschub)
14	0e	Umschaltung auf Text-Zeichensatz
15	0f	Blinken ein (80 Zeichen-Bildschim)
16		
17	11	CRSR abwärts
18	12	revers ein
19	13	home
20	14	delete
21		
22		
23		
24	18	Tabulator setzen/löschen
25		
26		
27	1b	ESCAPE
28		
29		
30		
31		

Dez	Hex	ASCII		DIN	
		Grafik	groß/klein	Grafik	groß/klein
32	20	SPACE	SPACE	SPACE	SPACE
33	21	!	!	!	!
34	22	"	"	"	"
35	23	#	#	#	#
36	24	$	$	$	$
37	25	%	%	%	%
38	26	&	&	&	&
39	27	'	'	'	'
40	28	(	(	(	(
41	29	)	)	)	)
42	2a	*	*	*	*
43	2b	+	+	+	+
44	2c	,	,	,	,
45	2d	–	–	–	–
46	2e	.	.	.	.
47	2f	/	/	/	/
48	30	0	0	0	0
49	31	1	1	1	1
50	32	2	2	2	2
51	33	3	3	3	3
52	34	4	4	4	4
53	35	5	5	5	5
54	36	6	6	6	6
55	37	7	7	7	7
56	38	8	8	8	8
57	39	9	9	9	9
58	3a	:	:	:	:
59	3b	;	;	;	;
60	3c	<	<	<	<
61	3d	=	=	=	=
62	3e	>	>	>	>
63	3f	?	?	?	?
64	40	@	@	§	§
65	41	A	a	A	a
66	42	B	b	B	b
67	43	C	c	C	c
68	44	D	d	D	d

Dez	Hex	ASCII Grafik	ASCII groß/klein	DIN Grafik	DIN groß/klein
69	45	E	e	E	e
70	46	F	f	F	f
71	47	G	g	G	g
72	48	H	h	H	h
73	49	I	i	I	i
74	4a	J	j	J	j
75	4b	K	k	K	k
76	4c	L	l	L	l
77	4d	M	m	M	m
78	4e	N	n	N	n
79	4f	O	o	O	o
80	50	P	p	P	p
81	51	Q	q	Q	q
82	52	R	r	R	r
83	53	S	s	S	s
84	54	T	t	T	t
85	55	U	u	U	u
86	56	V	v	V	v
87	57	W	w	W	w
88	58	X	x	X	x
89	59	Y	y	Y	y
90	5a	Z	z	Z	z
91	5b	[	[	[	[
92	5c	£	£	\	\
93	5d	]	]	]	]
94	5e	↑	↑	↑	↑
95	5f	←	←	–	–
96	60				
97	61		A		A
98	62		B		B
99	63		C		C
100	64		D		D
101	65		E		E
102	66		F		F
103	67		G		G
104	68		H		H
105	69		I		I
106	6a		J		J
107	6b		K		K
108	6c		L		L
109	6d		M		M
110	6e		N		N

Dez	Hex	ASCII Grafik	ASCII groß/klein	DIN Grafik	DIN groß/klein
111	6f		O		O
112	70		P		P
113	71		Q		Q
114	72		R		R
115	73		S		S
116	74		T		T
117	75		U		U
118	76		V		V
119	77		W		W
120	78		X		X
121	79		Y		Y
122	7a		Z		Z
123	7b				Ä
124	7c				Ö
125	7d				Ü
126	7e			π	π
127	7f				

Dez	Hex	Funktion
128	80	
129	81	orange
130	82	Unterstreichen aus
131		
132	84	HELP
133	85	f1
134	86	f3
135	87	f5
136	88	f7
137	89	f2
138	8a	f4
139	8b	f6
140	8c	f8
141	8d	SHIFT RETURN
142	8e	Umschaltung auf Grafik-Zeichensatz
143	8f	Blinken aus
144	90	Schwarz
145	91	Cursor hoch
146	92	Revers aus
147	93	CLEAR
148	94	INST
149	95	braun
150	96	hellrot
151	97	grau 1
152	98	grau 2
153	99	hellgrün
154	9a	hellblau
155	9b	grau 3
156	9c	pur
157	9d	Cursor links
158	9e	gelb
159	9f	türkis

Dez	Hex	ASCII Grafik	ASCII groß/klein	DIN Grafik groß/klein	DIN groß/klein
160	a0	SHIFT-Space		SHIFT-Space	
161	a1				
162	a2				
163	a3				
164	a4				
165	a5				
166	a6				
167	a7				
168	a8				
169	a9				
170	aa				
171	ab				
172	ac			é	é
173	ad			£	£
174	ae			è	è
175	af				
176	b0			@	@
177	b1			μ	μ
178	b2			à	à
179	b3			ù	ù
180	b4			â	â
181	b5			ê	ê
182	b6			î	î
183	b7			ô	ô
184	b8			û	û
185	b9			√	√
186	ba		↙	Σ	Σ
187	bb			Ä	ä
188	bc			Ö	ö
189	bd			Ü	ü
190	be			ß	ß
191	bf			^	^
192	c0				
193	c1	♠	A		A
194	c2		B		B
195	c3		C		C
196	c4		D		D
197	c5		E		E
198	c6		F		F
199	c7		G		G
200	c8		H		H
201	c9		I		I

Dez	Hex	ASCII Grafik	ASCII groß/klein	DIN Grafik	DIN groß/klein
202	ca		J		J
203	cb		K		K
204	cc		L		L
205	cd		M		M
206	ce		N		N
207	cf		O		O
208	d0		P		P
209	d1		Q		Q
210	d2		R		R
211	d3	♥	S		S
212	d4		T		T
213	d5		U		U
214	d6		V		V
215	d7		W		W
216	d8	♣	X		X
217	d9		Y		Y
218	da	♦	Z		Z
219	db				Ä
220	dc				Ö
221	dd				Ü
222	de	π		π	π
223	df			–	–
224	e0	SHIFT-Space		SHIFT-Space	
225	e1				
226	e2				
227	e3				
228	e4				
229	e5				
230	e6				
231	e7				
232	e8				
233	e9				
234	ea				
235	eb				
236	ec			é	é
237	ed			£	£
238	ee			è	è
239	ef			'	'
240	f0			@	@
241	f1			μ	μ
242	f2			à	à
243	f3			ù	ù

Dez	Hex	ASCII Grafik	ASCII groß/klein	DIN Grafik	DIN groß/klein
244	f4			â	â
245	f5			ê	ê
246	f6			↑	↑
247	f7			ô	ô
248	f8			û	û
249	f9			√	√
250	fa		✓	Σ	Σ
251	fb			Ä	ä
252	fc			Ö	ö
253	fd			Ü	ü
254	fe			ß	ß
255	ff	π		π	π

Standard ASCII-Code

HEX LSB	MSB BIN	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	Leerz.	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	ı	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	–	o	DEL

Umrechnung von hexadezimal nach dezimal

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Stichwortverzeichnis

von Larry Greenly u. a.

Commodores Originalbuch-Handbuch für Programmierer. Mehr brauchen Sie nicht, um den leistungsfähigen Commodore PC 128 schnell kennen zu lernen und direkt sicher für Ihre Aufgabenstellungen nutzen zu können.

880 Seiten, Best.-Nr. **3618** (1986)

Die SYBEX-Bibliothek

Commodore

DAS GROSSE COMMODORE BASIC HANDBUCH
von Michael Orkim – BASIC komplett für alle Commodore-Rechner von VC 20 bis C128. BASIC-Versionen 2.0, 3.5, 4.0, 7.0. Mit Tips für die Programmübertragung zwischen den einzelnen Modellen und für Befehls-Simulation sowie BASIC-Erweiterungen. 640 Seiten, Best.-Nr. **3615** (1986)

C 128 STARTEXTER
von Toni Schwaiger – Die Textverarbeitung der Spitzenklasse auch für professionelle Anwender mit dem Commodore C 128. Außergewöhnliche features, die den C 128 zum Textverarbeitungs-Star werden lassen – zum kleinen Preis. Diskette + Trainingsbuch (120 Seiten), Best.-Nr. **3415** (1986)

COMMODORE 128 STARDATEI
von Toni Schwaiger, dem Autor des Textverarbeitungs-Pakets Commodore 128 StarTexter. Ein leistungsfähiges und komfortables Dateiverwaltungs-Programm der Profiklasse mit Trainingsbuch, natürlich voll kompatibel zu StarTexter. Diskette + Trainingsbuch, Best.-Nr. **3420** (1987)

COMMODORE 128 STARPAINTER
von Heino Hansen und Elmar Sonnenschein. Das bedienerfreundliche Grafikprogramm der vielen Möglichkeiten, mit dem Sie professionelle Grafiken auf Ihrem C 128 erstellen. Den reibungslosen Einstieg ermöglicht das ausführliche Trainingsbuch. Diskette + Trainingsbuch, Best.-Nr. **3422** (1987)

COMMODORE 128 – ARBEITEN MIT TURBO PASCAL
von Karl-Hermann Rollke – Das SYBEX-Standardwerk „Arbeiten mit Turbo Pascal" wurde auf die für den C 128 gültige System- und Arbeitsumgebung abgestimmt und duch zusätzliche rechnerspezifische Informationen ergänzt. 296 Seiten, mit Abb., Best.-Nr. **3650** (1986)

COMMODORE 128 – ARBEITEN MIT dBASE II
von Michael A. Beisecker – Das Lehr- und Nachschlagewerk für die optimale Nutzung von dBASE II auf dem Commodore 128; der Leser wird anhand vieler Beispiele mit der Programmiersprache und deren Kommandos vertraut gemacht: Installation von dBASE II, Aufbau einer relationalen Datenbank, Arbeiten im interaktiven Befehlsmodus, Programmieren mit dBASE II, Editieren mit WordStar u.v.m. 280 Seiten, mit Abb., Best.-Nr. **3661** (1987)

COMMODORE 128 STARCOMM
von Jörg Jobst – Ein universelles Kommunikations- und Terminalprogramm zur Verbindung mit Mailboxen, größeren Hosts und Computern untereinander. Menügesteuerte Funktionswahl und Einstellung der Übertragungsparameter; Editieren und Korrigieren von Texten auf dem Bildschirm; Abspeichern von Dateien auf oder Abruf von Diskette. Voll kompatibel zum Textverarbeitungs-Programm **StarTexter.** Diskette + ausführliches Handbuch, Best.-Nr. **4007** (1986)

Assembler

PROGRAMMIERUNG DES 6502 mit 6510/65C02/65SC02
von Rodnay Zaks – Programmierung in Maschinensprache mit dem Mikroprozessor 6502 und anderen Mitgliedern der 65xx Familie, von den Grundkonzepten bis hin zu fortgeschrittenen Informationsstrukturen. 3. überarbeitete und erweiterte Ausgabe. 440 Seiten, 170 Abbildungen, Best.-Nr.: **3600** (1985)

FORTGESCHRITTENE 6502-PROGRAMMIERUNG
von Rodnay Zaks – hilft Ihnen, schwierige Probleme mit dem 6502 zu lösen, stellt Ihnen Maschinenroutinen zum Arbeiten mit einem Hobbyboard vor. 288 Seiten, 140 Abbildungen, Best.-Nr.: **3047** (1984)

PROGRAMMIERUNG DES Z80
von Rodnay Zaks – ein umfassendes Nachschlagewerk zum Z80-Mikroprozessor – jetzt in einer durch Lösungen ergänzten Ausgabe. 2., erweiterte Ausgabe. 640 Seiten, 176 Abbildungen, Best.-Nr.: **3099** (1985)

Z80 ANWENDUNGEN
von J. W. Coffron – vermittelt alle nötigen Anweisungen, um Peripherie-Bausteine mit dem Z80 zu steuern und individuelle Hardware-Lösungen zu realisieren. 296 Seiten, 204 Abbildungen, Best.-Nr.: **3037** (1984)

Systemsoftware

PROGRAMMIEREN MIT CP/M
von A. R. Miller – vermittelt die Feinheiten von CP/M und hilft, die Möglichkeiten dieses populären Betriebssystems zu erweitern. 424 Seiten, 97 Abb., Best.-Nr. **3077** (1985)

CP/M-HANDBUCH
von Rodnay Zaks – das Standardwerk über CP/M, das meistgebrauchte Betriebssystem für Mikrocomputer. Für Anfänger eine verständliche Einführung, für Fortgeschrittene ein umfassendes Nachschlagewerk über die CP/M-Versionen 2.2, 3.0 und CCP/M-86 sowie MP/M., 2. überarbeitete Ausgabe. 356 Seiten, 56 Abbildungen, Best.-Nr.: **3053** (1984)

SYBEX

EDASS 128
Diskette zum
C 128 Assembler-Kurs

SYBEX

SYBEX

25 1889

***Bitte sofort Sicherungskopie anfertigen
und Original-Diskette
gut geschützt aufbewahren!***

SYBEX

COMMODORE 128
Assembler-Kurs

Das vorliegende Buch führt den an der Programmierung in Maschinensprache auf dem C128 interessierten Leser Schritt für Schritt kompetent und leicht verständlich in die Assemblersprache ein. Die beiliegende Diskette enthält einen äußerst leistungsfähigen Makro-Assembler mit bedienerfreundlichem Editor und Reassembler sowie einen Maschinensprache-Monitor. Alle im Begleitbuch erläuterten Programmbeispiele nutzen die Betriebssystemumgebung des C128 und dessen Hardware. Der Kurstext wird durch Übungsaufgaben mit Lösungen ergänzt.

Für jeden Besitzer eines Commodore 128 stellt das Buch zusammen mit der zugehörigen Diskette nicht nur einen wertvollen Assemblerkurs, sondern auch ein komplettes Entwicklungspaket für den Entwurf und den Test selbstgeschriebener Assemblerprogramme dar.

Aus dem Inhalt:

- Assemblersprache, was ist das?
- Ein erstes Programm
- Daten transportieren und manipulieren
- Adressierungsarten
- Arithmetik im Assemblerprogramm
- Bitmanipulationen
- Der Maschinensprache-Monitor
- Makroassemblierung
- Der Reassembler
- EDASS-Rechenfunktionen
- Tips zur Programmerstellung

Die Programmdiskette enthält zusätzlich noch nützliche Dienstprogramme und Makros zur 16-bit-Arithmetik sowie zur strukturierten Programmierung.

ISB N 3-88745-522-3

DM 75,—
sFr 69,—
öS 585,—

9 783887 455224