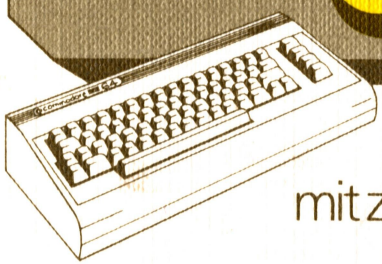
H. Sterner T. Tutughamiarsa

MASCHINEN-SPRACHE AUF DEM COMMODORE 64



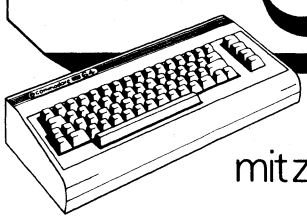
Eine Einführung
mit zahlreichen Anwendungs-
beispielen

iwt

H. Sterner/T. Tutughamiarsa
Maschinensprache auf dem Commodore 64

H. Sterner T. Tutughamiarsa

MASCHINEN-SPRACHE AUF DEM COMMODORE 64



Eine Einführung
mit zahlreichen Anwendungs-
beispielen

iwt

ISBN 3-88322-047-7
1. Auflage 1985

Herstellung: Freiburger Graphische Betriebe, Freiburg
Umschlaggestaltung: Kaselow und Partner, München

Vorwort

Mit dem Thema "Maschinensprache" wird man spätestens dann konfrontiert, wenn man feststellt, wie bescheiden die Möglichkeiten des im Commodore 64 eingebauten BASIC-Interpreters sind. Dies gilt sowohl in Hinsicht auf den Komfort (denken Sie daran, mit welchem Aufwand Sie einen einzigen Grafik-Punkt programmieren müssen!) als auch in Bezug auf die Geschwindigkeit, mit der z.B. eine Grafikdarstellung in BASIC auf den Bildschirm zu bringen ist.

Eine Möglichkeit, diese Schwächen des Commodore-BASIC zu umgehen, besteht natürlich darin, daß man sich eine der verschiedenen BASIC- oder Grafik-Erweiterungen zulegt. Für denjenigen jedoch, der seine ganz spezifischen Probleme lösen will, gibt es nur die Möglichkeit, selbst in die Maschinensprache-Programmierung "einzusteigen". Dabei möchte das vorliegende Buch behilflich sein.

Der 6510-Prozessor des Commodore 64 "versteht" ca. 50 Befehle, von denen die meisten verschiedene Adressierungsarten besitzen, so daß sich insgesamt etwa 150 verschiedene Befehls-Möglichkeiten ergeben. Es ist offensichtlich wenig sinnvoll, diesen Befehlssatz ohne Anwendungsbeispiele zunächst nur "auf dem Trockenen" zu besprechen.

Deshalb kommt dieses Buch nach einer kurzen Darstellung einiger notwendiger Grundlagen gleich zur Sache: anhand von kurzen Beispiel-Programmen, die aufeinander aufbauen, werden zunächst einige der wichtigsten Befehle illustriert. Weitere Befehle werden dann eingeführt, wenn sie für die Problemlösung erforderlich sind. Auf diese Weise erarbeitet man sich den "Wortschatz" der Maschinensprache in dem Zusammenhang, in dem er tatsächlich benötigt wird.

Die Darstellung endet jedoch nicht mit den einführenden Programm-Beispielen: im Mittelpunkt des Buches steht ein Paket von 16-Bit-Routinen, die ein 8-Bit-Prozessor - wie der 6510 - bekanntlich nicht direkt ausführen kann, die jedoch z.B. für die Grafik-Programmierung sehr wichtig sind.

Weiterhin werden einige häufig verwendete interne ROM-Routinen des Commodore 64 beschrieben, die sich auch in eigenen Programmen verwenden lassen.

Abschließend wird eine Gesamtübersicht über die Adressierungsarten des 6510 gegeben.

Ein ausführlicher Anhang und ein Stichwortverzeichnis erlauben einen schnellen Zugriff auf die Beschreibung der einzelnen Befehle und sonstige Stichwörter.

Demjenigen, der keinen Assembler besitzt, erleichtern BASIC-Ladeprogramme die Eingabe. Die Programme stehen jedoch auch zusätzlich auf Kassette oder Diskette zur Verfügung und können direkt beim Verlag bestellt werden.

Außerdem finden Sie im Anhang ein Disassembler-Programm, das zusätzlich die Eingabe von Maschinenprogrammen direkt im üblichen Hexadezimalcode erlaubt.

Viel Spaß und Erfolg beim Einstieg in die Programmierung in Maschinensprache wünschen Ihnen

die Autoren!

Inhalt

Vorwort 5

Einleitung 11

1. Schematische Darstellung des Mikroprozessors 6510 13
   - 1.1 Akkumulator 16
   - 1.2 Index-Register X und Y 16
   - 1.3 Programmzähler 16
   - 1.4 Stapelzeiger 16
   - 1.5 Prozessor-Status-Register 17

2. Algorithmen 19
   - 2.1 Hauptaufgaben einer Programmiersprache ... 22
   - 2.2 Grundlegende Elemente zur Darstellung von Flußdiagrammen 22

3. Zahlensysteme 27
   - 3.1 Das Binär-System 29
     - 3.1.1 Die Addition mit Binärzahlen 30
     - 3.1.2 Zahlen mit Vorzeichen 30
       - 3.1.2.1 Das Einerkomplement 31
       - 3.1.2.2 Das Zweierkomplement 31
     - 3.1.3 Übertrag und Überlauf 32
     - 3.1.4 High- und Low-Byte 35
   - 3.2 Das Hexadezimal-System 35

4. Wie schreibt man ein Programm in Assembler? ... 39
   - 4.1 Addition zweier Zahlen 41
   - 4.2 Addition mehrerer Zahlen 46
   - 4.3 16-Bit-Addition 52

5. Unterprogramme 59
   - 5.1 Parameterübergabe 64
     - 5.1.1 Parameterübergabe durch Register .. 65
     - 5.1.2 Parameterübergabe durch den Stack . 66
     - 5.1.3 Parameterübergabe durch vereinbarte Speicher 70
     - 5.1.4 Direkte Parameterübergabe 73

6. 16-Bit-Simulationen 75
   - 6.1 Häufig verwendete 16-Bit-Routinen 80
   - 6.2 Der 16-Bit-Interpreter 101

7. Grafik-Anwendungen der Assembler-Routinen 107
7.1 16-Bit-Routinen zur hochauflösenden Grafik 109
7.1.1 Grafik einschalten 109
7.1.2 Grafik ausschalten 110
7.1.3 Grafikspeicher löschen 110
7.1.4 Hintergrundfarbe setzen 112
7.1.5 Adressberechnung für Grafik-speicher 112
7.1.6 Grafikpunkt setzen/löschen 117
7.1.7 Zeichnen und Löschen von Geraden .. 120
7.2 16-Bit-Routinen zur Sprite-Programmierung 126
7.2.1 Erstellen von Sprites 129
7.2.2 Sprite-Positionierung 133
7.3 Weitere Routinen zur Sprite-Programmierung 134
7.3.1 Ein- und Ausschalten von Sprites .. 134
7.3.2 Bestimmung der Sprite-Farbe 136
7.3.3 Bestimmung der Anfangsadresse von Sprites 137
7.4 Anwendungsbeispiele zur Benutzung der Sprite- und Grafik-Routinen 138
7.4.1 Beispiel zur Benutzung der Sprite-Routinen 139
7.4.2 Beispiel zur Benutzung der Grafik-Routinen 140

8. Verwendung von ROM-Routinen 141
8.1 Parameterübergabe von BASIC-Programmen aus 143
8.2 Weitere Beispiele mit ROM-Routinen 147

9. Kurze Darstellung der in den Programmen nicht verwendeten Befehle 153
9.1 Dezimal-Arithmetik 156
9.2 Programmunterbrechungen (Interrupts) 156

10. Adressierungsarten 159
10.1 Nicht indizierte Adressierungsarten 161
10.1.1 Implizite Adressierung 161
10.1.2 Unmittelbare Adressierung 161
10.1.3 Absolute Adressierung 162
10.1.4 Zero Page Adressierung 162
10.1.5 Relative Adressierung 162
10.2 Indizierte Adressierungsarten 163
10.2.1 Absolut indizierte Adressierung . 164
10.2.2 Zero Page Adressierung 164
10.2.3 Indirekte Adressierung 164
10.2.4 Indiziert-indirekte Adressierung 165
10.2.5 Indirekt-indizierte Adressierung 165
10.2.6 Indirekt absolute Adressierung .. 166

Anhang 167
Alphabetische Kurzübersicht der 6510-Befehle . 169
Alphabetische Übersicht der 6510-Befehle und Adressierungsarten 172
6510-Befehlsliste in hexadezimaler Reihenfolge 201
Klassifikation der 6510-Befehle 205
Hexadezimalcode-Eingabe/Disassembler 206
BASIC-Ladeprogramm-Generator 214
Listing der 16-Bit-Routinen 216
BASIC-Ladeprogramm für die 16-Bit-Routinen ... 234

Literaturverzeichnis 239

Register 240

Einleitung

Was ist ein Mikroprozessor?

Ein Mikroprozessor ist - kurz gesagt - ein Baustein zur Steuerung und Verarbeitung von Daten. Es handelt sich dabei um eine hochintegrierte Schaltung, bei der Tausende von Schaltfunktionen auf einer Fläche von nur wenigen Quadratmillimetern untergebracht sind.

Ein solcher Mikroprozessor stellt die Zentraleinheit oder CPU (Central Processing Unit) eines Mikrocomputers dar. Im Unterschied zu größeren Computern besteht die Zentraleinheit eines Mikrocomputers also aus nur einem Chip (= integrierte Schaltung).

Warum in Maschinensprache programmieren?

Während der Mikroprozessor in den meisten anderen Geräten, in denen er zur Steuerung eingesetzt wird, nur einmal für eine bestimmte Aufgabe programmiert wird, ist der in einem Mikrocomputer verwendete Mikroprozessor dazu bestimmt, laufend neu programmiert zu werden, um so die unterschiedlichsten Aufgaben zu lösen.

Diese Programmierung eines Mikroprozessors erfolgt mit Hilfe eines Befehlssatzes, der aus den Befehlen besteht, die der betreffende Mikroprozessor hardwaremäßig "versteht", d.h. die von ihm direkt ausgeführt werden können. Diese Befehle sind die Sprachelemente der Maschinensprache. Da dies die einzige Programmiersprache ist, die ein Mikroprozessor versteht, müssen die Befehle irgend einer "höheren" oder "problemorientierten" Sprache (wie z.B. BASIC oder Pascal) also vor der Programmausführung jeweils in die Maschinensprache übersetzt werden.

Das hierfür zuständige "Übersetzer-Programm" wird als "Compiler" bezeichnet, wenn das vorliegende Programm vor der Ausführung vollständig übersetzt wird. Von einem "Interpreter" spricht man dann, wenn die einzelnen Anweisungen eines Programmes gleich nach dem Übersetzen ausgeführt werden. Ein solches "interpretiertes" Programm läuft natürlich erheblich langsamer, da jede Anweisung - z.B. auch innerhalb einer Schleife - immer wieder neu übersetzt wird. Der Vorteil besteht

darin, daß man das Programm nach einer Änderung sofort wieder neu starten kann und nicht erst die vollständige Übersetzung abwarten muß.

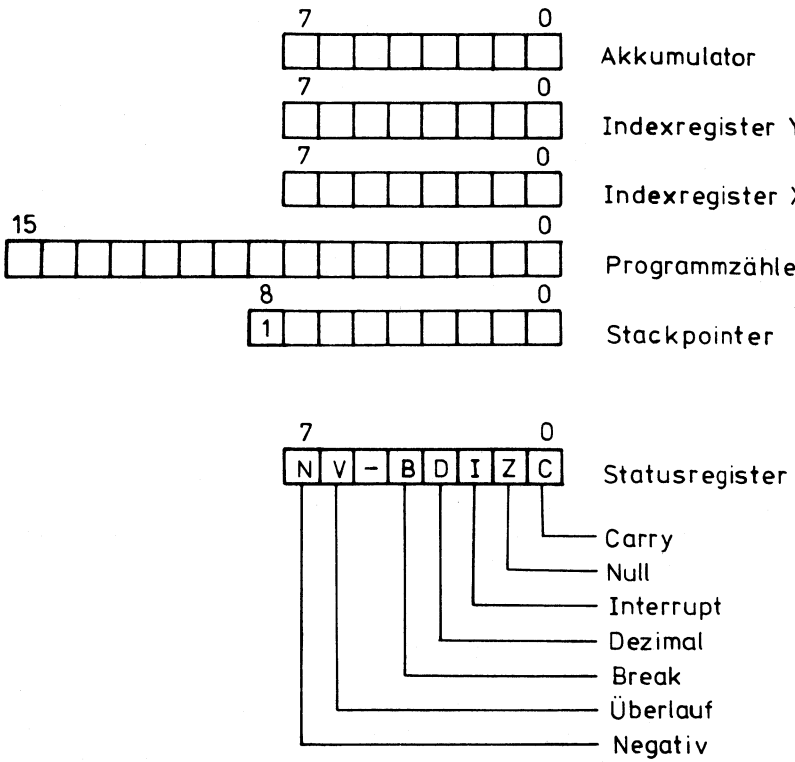
In jedem Fall läuft natürlich ein Programm in Maschinensprache erheblich schneller ab, als in irgendeiner anderen Programmiersprache. Allerdings hat dieser Vorteil auch seinen Preis: das Programmieren in Maschinensprache ist nicht so leicht zu erlernen, wie das Programmieren in vielen anderen, höheren Programmiersprachen. Außerdem ist der Aufwand zur Erstellung eines Programmes im allgemeinen sehr viel größer, als bei der Programmierung in anderen Sprachen. Deshalb schließt man auch vielfach einen Kompromiß und programmiert nur besonders zeitkritische Programmteile in der Maschinensprache, die dann z.B. in BASIC-Programmen durch einen entsprechenden Aufruf verwendet werden können.

Ein typisches Beispiel beim Commodore 64 bietet die Programmierung der Grafik, die durch keine speziellen BASIC-Befehle unterstützt wird, so daß allein zum Setzen eines einzigen Grafikpunktes ein mehrzeiliges Unterprogramm erforderlich ist, was schon seine Zeit kostet! An bewegte hochauflösende Grafikdarstellungen ist in BASIC also erst recht nicht zu denken. Hier gibt es einfach keine andere Möglichkeit, als solche Probleme in Maschinensprache zu lösen. Als Anreiz mag vielleicht ein Beispiel zum Geschwindigkeitsvorteil, den ein Programm in Maschinensprache bietet, dienen: während das Löschen des Grafikspeichers in BASIC gut 30 Sekunden dauert, passiert dies in Maschinensprache in einem kaum wahrnehmbaren Sekundenbruchteil!

1 | Schematische Darstellung des Mikroprozessors 6510

1. Schematische Darstellung des Mikroprozessors 6510

Es ist jedoch nicht unbedingt notwendig, das "Innenleben" eines Mikroprozessors mit seinen Tausenden von Funktionselementen im einzelnen kennenzulernen, um den Mikroprozessor zu programmieren. Um zu verstehen, was sich innerhalb des Mikroprozessors abspielt, reicht schon eine schematische Darstellung des Aufbaus und der Funktion dieser komplexen integrierten Schaltung aus. Es genügt schon ein einfaches Programmiermodell des Mikroprozessors, wie es in der folgenden Abbildung angedeutet ist.



Wie Sie der Abbildung entnehmen können, sind im Grunde nur sehr wenige Register (= Speicher innerhalb eines Prozessors) für den Programmierer von Bedeutung. Im einzelnen handelt es sich dabei um folgende Register:

1.1 Akkumulator

Der Akkumulator (A) ist das eigentliche Arbeitsregister des Prozessors, in dem die Ergebnisse aller arithmetischen und logischen Verknüpfungen gespeichert werden.

1.2 Index-Register X und Y

Die Index-Register X und Y dienen auch dazu, Daten für arithmetische oder logische Verknüpfungen aufzunehmen, doch besteht ihre wichtigste Aufgabe darin, die Speicheradressierung zu unterstützen. Den Inhalt dieser Register kann man zum Inhalt eines jeden anderen Speichers addieren. Auf diese Weise erhält man einen Versatz oder "Offset", wodurch ein effektiver Zugriff auf Tabellen-Daten möglich ist. Das X-Register ist etwas flexibler, da es Zero Page Operationen zuläßt, die das Y-Register mit Ausnahme der LDX- und STX-Befehle nicht hat. (Nähere Erläuterungen dazu folgen später, insbesonders in dem Kapitel über Adressierungsarten).

1.3 Programmzähler

Der Programmzähler (PC = Program Counter) ist das einzige 16-bit-Register des Prozessors. Er enthält die jeweils aktuelle Adresse des zu bearbeitenden Befehls innerhalb eines Programmes.

1.4 Stapelzeiger

Der Stapelzeiger (S = Stack pointer) dient zur Aufnahme eines Zeigers für den Stapelspeicher (stack). Dieser Speicherbereich ist so organisiert, daß die zuletzt gespeicherten Daten zuerst wieder ausgelesen werden können (LIFO = Last In, First Out). Dieser Stack wird vom Prozessor sehr oft (ohne besondere Anweisungen

des Programmierers) verwendet. Andererseits kann jedoch der Programmierer auch direkt die Vorteile dieser besonderen Speicher-Organisation ausnutzen, wie z.B. bei der Parameter-Übergabe zwischen Haupt- und Unterprogramm. Der Stack-Pointer enthält zusätzlich ein 9. Bit, das immer gesetzt ist. Dadurch hat dieses Register einen festen Adressbereich, nämlich von 1FF bis 100.

1.5 P r o z e s s o r - S t a t u s - R e g i s t e r

Das Prozessor-Status-Register (P) enthält Informationen über den aktuellen Zustand des Prozessors, um z.B. den Überlauf nach einer Addition festzuhalten. Von den 8 möglichen werden nur 7 Bits benutzt, die jeweils einzeln angesprochen werden können. Diese Bits - auch als Flags oder Flaggen bezeichnet - werden bei jeder logischen oder arithmetischen Operation aktualisiert. Dadurch ist es möglich, den jeweiligen Zustand des Prozessors nach bestimmten Operationen abzufragen. Die Flags haben im einzelnen folgende Funktionen:

Das N-Flag oder S-Flag (Negative flag oder Sign flag, = Bit 7) dient zum Festhalten eines Vorzeichens.

Das V-Flag (oVerflow flag, = Bit 6) wird gesetzt, wenn ein Überlauf in die Vorzeichenposition erfolgt. Es braucht nur beim Rechnen mit negativen Zahlen berücksichtigt zu werden.

Das B-Flag (Break flag, = Bit 4) wird vom Prozessor gesetzt, wenn eine Software-Unterbrechung stattgefunden hat. Es handelt sich um eine Besonderheit des 6510, die eine softwaregesteuerte Programmunterbrechung (software interrupt) erlaubt. Findet eine solche Unterbrechung statt, so wird dieses Flag gesetzt, so daß man zwischen Software- und Hardware-Interrupt unterscheiden kann (vgl. Kap. 9).

Das D-Flag (Decimal flag, = Bit 3) weist ebenfalls auf eine Besonderheit des 6510 hin. Normalerweise rechnet der Prozessor im Binärsystem. Ist jedoch dieses Flag im Programm gesetzt, so werden die nachfolgenden arithmetischen Operationen im Dezimalsystem (im BCD-Code, d.h. Binary Coded Decimal) ausgeführt (vgl. Kap. 9).

Das I-Flag (Interrupt flag oder Interrupt mask, = Bit 2) zeigt an, ob eine maskierte Unterbrechung möglich ist (vgl. Kap. 9).

Das Z-Flag (Zero flag, = Bit 1) wird von einer ganzen Reihe von Befehlen gesetzt, wenn der Wert Null auftritt, z.B. dann, wenn das Ergebnis einer Addition gleich Null ist oder ein bestimmter Speicher bzw. ein Register mit Null geladen (oder der Wert Null aus einem Speicher bzw. Register gelesen) wird.

Das C-Flag (Carry flag, = Bit 0) wird gesetzt, wenn nach einer Operation ein Übertrag entsteht.

2 | Algorithmen

2. Algorithmen

Sofern ein Problem nicht gerade extrem einfach ist, empfiehlt es sich, vor der Programmierung den Lösungsweg in übersichtlicher Form darzustellen. Dazu eignet sich am besten eine grafische Darstellung, die den Ablauf der einzelnen Schritte veranschaulicht. Eine solche Darstellung nennt man dementsprechend Ablauf- oder Flußdiagramm.

Damit haben wir auch gleich einen wichtigen Begriff anhand eines Beispiels erklärt, nämlich den Begriff Algorithmus:

Ein Flußdiagramm ist nämlich nichts anderes als eine grafische Darstellung eines Algorithmus, also einer Folge von Anweisungen, die (wenn sie entsprechend detailliert formuliert sind) es ermöglichen, ein bestimmtes Problem zu lösen.

Die Darstellung der einzelnen Schritte eines Algorithmus hängt natürlich ganz davon ab, für wen die Anweisungen bestimmt sind. Will man die Lösung einer Aufgabe einem Computer übertragen, so kann man dem Computer nicht direkt ein Flußdiagramm eingeben, sondern man muß vielmehr den Algorithmus in eine andere Form bringen, indem man ihn als Programm in einer Programmiersprache formuliert, die der Computer "verstehen" kann. Ein Programm ist also - ebenso wie ein Flußdiagramm - eine bestimmte Darstellung eines Algorithmus.

Fassen wir noch einmal zusammen:

Ein Algorithmus ist eine Folge von Anweisungen, die in einer bestimmten Reihenfolge abgearbeitet werden müssen, um zu einem vorgegebenen Ziel zu gelangen.

Die Darstellung eines Algorithmus kann auf verschiedene Weise erfolgen, z.B. grafisch in Form eines Flußdiagramms.

Für den Computer muß ein Algorithmus jedoch mit den Mitteln einer Programmiersprache formuliert werden, die der Computer "versteht", d.h. deren Anweisungen er ausführen kann.

Zur Erstellung eines Programms gehören also folgende Schritte:

1. Analyse des Problems (wie läßt sich das Problem zergliedern?).
2. Erstellung eines Programmablaufplans (Flußdiagramm).
3. "Übersetzung" in die Programmiersprache.

2.1 Hauptaufgaben einer Programmiersprache

Die Hauptaufgaben einer Programmiersprache lassen sich unter den Begriffen
- Eingabe,
- Ausgabe,
- Zuordnung,
- Arithmetik (Rechenoperationen) und
- Verzweigung

zusammenfassen.

Da jede Programmiersprache über entsprechende Sprachmittel verfügt, könnte man grundsätzlich jedes Problem, für dessen Lösung sich ein Algorithmus angeben läßt, in eine beliebige Programmiersprache übertragen. Praktisch sieht es jedoch so aus, daß sich bestimmte Probleme in einer bestimmten Sprache besser darstellen lassen als in einer anderen. Dies erklärt auch die Vielzahl der existierenden Programmiersprachen.

2.2 Grundlegende Elemente zur Darstellung von Flußdiagrammen

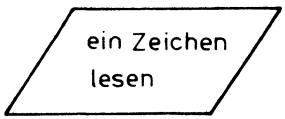
Im folgenden sollen die wichtigsten grafischen Elemente zur Darstellung von Flußdiagrammen vorgestellt werden.

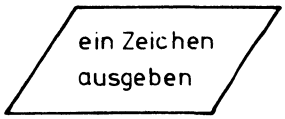
Allgemein besteht ein Flußdiagramm aus einer Anzahl von grafischen Symbolen (z.B. Rechtecken), in die in freier Formulierung die einzelnen Anweisungen eines Algorithmus eingetragen werden. Der Programmablauf wird durch entsprechende Verbindungslinien gekennzeichnet, wobei der Hauptfluß des Programms von oben nach unten erfolgt. Abweichende Richtungsverläufe werden durch Pfeile markiert.

Ein- und Ausgabe

Als Symbol für eine Ein- oder Ausgabe verwendet man ein Parallelogramm, das die notwendigen Angaben über die Ein- bzw. Ausgabewerte enthält:

Beispiel:

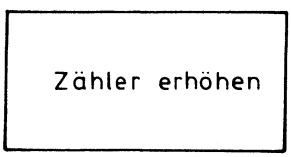




Zuordnung und allgemeine Operationen (Arithmetik)

Das allgemeine Symbol für beliebige Operationen ist ein Rechteck:

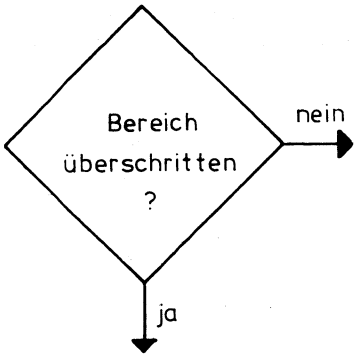
Beispiel:



Verzweigung

Verzweigungen ermöglichen es, je nach Ausgang einer Entscheidung, den Programmlauf an verschiedenen Stellen fortzusetzen:

Beispiel:



Programmanfang- und Ende

Programmanfang- und Ende werden durch das folgende Symbol gekennzeichnet. Eine solche Kennzeichnung mag auf den ersten Blick überflüssig erscheinen, ist jedoch zumindest bei umfangreicheren Flußdiagrammen notwendig.

Beispiel:

Anschlussstellen

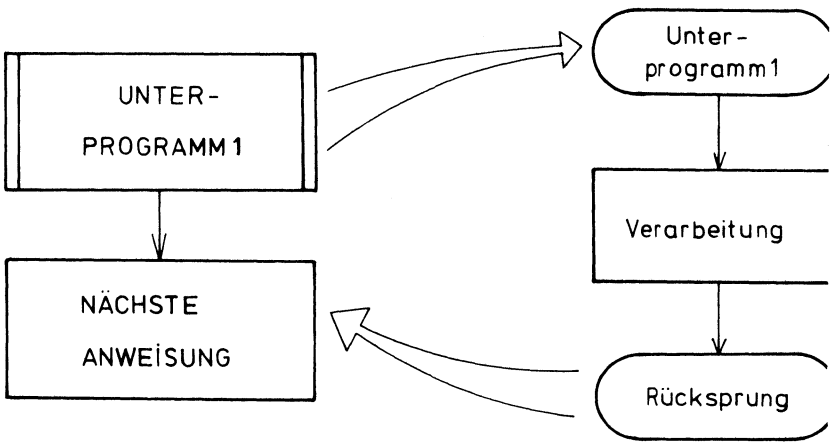
Nur kürzere Flußdiagramme lassen sich auf einer Seite unterbringen. Die "Fortsetzungen" eines Programms werden durch Kreis-Symbole (am jeweiligen Programmende- bzw. Fortsetzungsstück) dargestellt, die eine beliebige Kennzeichnung enthalten (z.B. Zahlen).

Beispiel:

Unterprogramme

Bei der Entwicklung von Flußdiagrammen empfiehlt es sich, so vorzugehen, daß man das zu lösende Problem schrittweise verfeinert, d.h. man beginnt mit einer groben Skizze des Programmablaufs und entwickelt nach und nach die einzelnen Abschnitte im Detail. Für einen modularen Programmaufbau sind deshalb Unterprogramme wichtig, die eine übersichtliche Struktur des Hauptprogrammes erlauben.

Beispiel:



3 | Zahlensysteme

3. Zahlensysteme

Die beiden wichtigsten Zahlensysteme bei der Programmierung sind das Binär- und das Hexadezimalsystem. Der Grund dafür liegt in der Hardware des Computers.

Da jeder Speicherplatz (Byte) in 8 Einheiten (Bits) unterteilt ist, die jeweils nur zwei Zustände annehmen können, bietet sich offensichtlich das Binärsystem zur Beschreibung dieser Zustände an.

3.1 Das Binär-System

Das binäre oder duale Zahlensystem verwendet nur die Ziffern 0 und 1.

Genau wie im Dezimalsytem bestimmt auch im Binärsystem die Stelle innerhalb einer Zahl den Wert einer Ziffer (z.B. dezimal 296 = 1*6 + 10*9 + 100*2).

Im Dezimalsystem werden die Ziffern entsprechend ihrer Position in der Zahl mit Potenzen von 10 multipliziert (dabei geht man von rechts nach links vor!): 10↑0 = 1, 10↑1 = 10, 10↑2 = 100 usw.

Im Binärsystem werden stattdessen Potenzen von 2 verwendet: 2↑0 = 1, 2↑1 = 2, 2↑2 = 4, 2↑3 = 8 usw. Die Binärzahl 1101 wird also folgendermaßen in eine Dezimalzahl umgerechnet:

```
1101 = 1*2↑3 + 1*2↑2 + 0*2↑1 + 1*2↑0
     = 1*8   + 1*4   + 0*2   + 1*1
     = 13 (dezimal)
```

Da ein Speicherplatz unseres Computers aus 8 Bits besteht, die alle den Wert 0 oder 1 haben können, ist der kleinste Wert, den ein Speicher aufnehmen kann:

00000000 (binär) = 0

und der größte:

```
11111111 (binär) = 1*2↑7 + 1*2↑6 + 1*2↑5 + 1*2↑4
                 + 1*2↑3 + 1*2↑2 + 1*2↑1 + 1*2↑0
                 = 255  (dezimal)
```

Aus diesem Grund kann ein Speicher nur Werte bis zu 255 (dezimal) enthalten.

3.1.1 Die Addition mit Binärzahlen

Für das Addieren im Binärsystem gelten folgende Regeln:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 Übertrag 1 (also: 10)

Beispiel:

```
  5 (dezimal)        1001 (binär)
+ 6 (dezimal)      + 1010 (binär)
-------------      --------------
 11 (dezimal)       10011 (binär)
```

Die Addition erfolgt wie im Dezimalsystem von rechts nach links: 1+0=1, 0+1=1, 0+0=0 und 1+1=0 Übertrag = 1, wobei der Übertrag zur nächsthöheren Stelle hinzuaddiert wird (in diesem Fall Bit 4 bzw. Zehnerstelle im Dezimalsystem).

3.1.2 Zahlen mit Vorzeichen

In der Praxis hat man es aber nicht nur mit positiven, sondern auch mit negativen Zahlen zu tun. Wie stellt man aber Zahlen im Binärsystem dar, die ein Vorzeichen - Minuszeichen - haben? Üblicherweise kennzeichnet man eine positive Zahl ja mit einem Plus- (+) und eine negative Zahl mit einem Minus- (-) Zeichen vor der Zahl. Im Binärsystem wird stattdessen im allgemeinen das höchstwertige Bit (MSB, Most Significant Bit)

zur Kennzeichnung des Vorzeichens benutzt. Dabei wird MSB = 0 für positive und MSB = 1 für negative Zahlen verwendet. Deshalb kann man unter Berücksichtigung des Vorzeichen-Bits in einem 8-Bit-Code also nur positive Zahlen bis +127 und negative bis -128 darstellen, insgesamt soviele, wie ohne Vorzeichen-Markierung. - Ein Beispiel dazu:

Die Dezimalzahlen +18 bzw. -18 werden binär - mit Berücksichtigung des Vorzeichens - so dargestellt:

```
+18 (dezimal) = 00010010
                ↑
                Vorzeichen-Bit

-18 (dezimal) = 11101110
                ↑
                Vorzeichen-Bit
```

Diese Form der Darstellung nennt man Zweierkomplement. Zunächst aber wollen wir uns mit dem Einerkomplement beschäftigen.

3.1.2.1 Das Einerkomplement

Die Darstellung positiver Zahlen als Einerkomplement unterscheidet sich nicht von der schon bekannten Binärschreibweise ohne Vorzeichen. Negative Zahlen werden hierbei durch Invertieren (Umkehren) der einzelnen Bits dargestellt, so wird z.B. aus 00010010 (dezimal +18) durch Invertieren 11101101 (dezimal -18). Sie sehen, daß sich auch hier bei einer Vorzeichenänderung das höchstwertige Bit (MSB) ändert.

3.1.2.2 Das Zweierkomplement

Bei den positiven Zahlen gibt es keinen Unterschied zum Einerkomplement. Anders sieht es bei den negativen Zahlen aus: diese werden jetzt so dargestellt, daß man zunächst das Einerkomplement bildet und dazu 1 addiert. - Ein Beispiel hierzu: Nehmen wir wieder die Dezimalzahl -18, so erhalten wir auf folgende Weise die entsprechende Binärdarstellung als Zweierkomplement:

```
18 (dezimal) = 00010010

               11101101 (-18 als Einerkomplement)
             +        1
             ----------
               11101110 (-18 als Zweierkomplement)
```

Im folgenden werden Zahlen mit Vorzeichen immer als Zweierkomplement dargestellt.

3.1.3 Übertrag und Überlauf

Werden Zahlen durch 8 Bits, also ein Byte, dargestellt, wie es bei unserem Mikroprozessor in der Regel der Fall ist, können Additionen nur bis zum 7. Bit durchgeführt werden. Was passiert aber, wenn durch Addition des 7. Bits ein Übertrag entsteht? Würde dieser Fall nicht berücksichtigt, erhielte man in manchen Fällen ein falsches Ergebnis. Um das Ergebnis bei einem Übertrag korrigieren zu können, wird das Übertrags- oder Carry-Bit auf den Wert 1 gesetzt.

Beispiel:

```
  130 (dezimal)        10000010 (binär)
+ 155 (dezimal)      + 10011011 (binär)
---------------      ------------------
  285 (dezimal)        00011101 (binär) Übertrag=1
```

Dieser Übertrag entspricht Bit Nr. 9, also dem Wert 256 (dezimal). Das Ergebnis der binären Addition ohne Berücksichtigung des Übertrags beträgt 29 (dezimal). Durch Addition von 256 (dezimal) ergibt sich die tatsächliche Summe von 285 (dezimal).

Das Überlauf- oder Overflow-Bit wird gesetzt, wenn sich der Wert des 7. Bits ändert, d.h. wenn ein Vorzeichenwechsel auftritt. Würde man diese Situation nicht berücksichtigen, könnten falsche Ergebnisse, entstehen, wenn sich ein Vorzeichen ändert.

Beispiel:

```
  85 (dezimal)        01010101 (binär)
+ 49 (dezimal)      + 00110001 (binär)
--------------      ------------------
 134 (dezimal)        10000110 (binär) Überlauf = 1
```

In diesem Beispiel haben wir zwei positive Zahlen addiert und erwarten demnach auch ein positives Ergebnis. Wenn wir aber die Zahlen als Zweierkomplement betrachten, würde das Ergebnis eine negative Zahl darstellen, weil das 7. Bit gleich 1 ist. Da jedoch das Überlauf-Bit gesetzt wurde, wissen wir, das in diesem Fall das 7. Bit kein negatives Vorzeichen darstellen soll und können das Ergebnis entsprechend korrigieren.

In den folgenden Beispielen werden alle Möglichkeiten gezeigt, die sich bei der Berücksichtigung des Übertrag- (C) und Überlauf-Bits (V) ergeben können.

Addition zweier positiver Zahlen:

```
  17              00010001
+  9            + 00001001
----            ----------
  26              00011010          C = 0     V = 0
```

Da sowohl C als auch V gleich Null sind, braucht das Ergebnis nicht korrigiert zu werden.

```
  87              01010111
+ 66            + 01000010
----            ----------
 153              10011001          C = 0     V = 1
```

Da das 7. Bit des Ergebnisses gleich 1 ist, würde es ohne Korrektur als negativ angesehen. Durch Abfrage des Overflow-Bits kann das Ergebnis korrigiert werden. (Die Erklärung dazu folgt später).

Addition von positiver und negativer Zahl:

```
   33            00100001
+ (-2)         + 11111110
------         ----------
   31            00011111          C = 1     V = 0


   10            00001010
+ (-17)        + 11101111
-------        ----------
   - 7           11111001          C = 0     V = 0
```

Wenn die Operanden verschiedene Vorzeichen haben (hier: +33 und -2 bzw. +10 und -17), ist das Ergebnis der Addition ohne Berücksichtigung des Carry- bzw. Overflow-Bit immer richtig. Wie sie sehen, wird die Subtraktion durch die Verwendung des Zweierkomplements auf die Addition zurückgeführt (Komplementaddition).

Addition zweier negativer Zahlen:

```
 (-12)           11110100
+ (-5)         + 11111011
------         ----------
 - 17            11101111          C = 1     V = 0


  (-80)          10110000
+ (-99)        + 10011101
-------        ----------
  -179            01001101          C = 1     V = 1
```

Während im ersten Beispiel das Ergebnis (trotz des gesetzen Carry-Bits) ohne Korrektur richtig ist, muß es im zweiten Fall korrigiert werden, weil das Überlauf-Bit gesetzt ist.

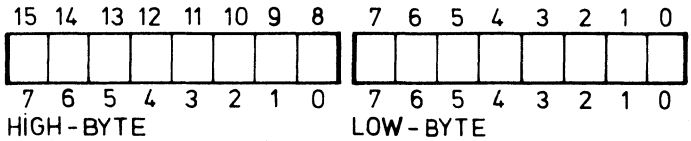
Allgemein kann man diesen Beispielen folgendes entnehmen:

Bei der Addition von Zahlen ohne Vorzeichen (also von positiven Zahlen) ist eine Korrektur des Ergebnisses in Abhängigkeit vom Zustand des Übertrag- (Carry-) Bit notwendig.

Bei der Addition von Zahlen mit Vorzeichen muß der Zustand des Überlauf- (Overflow-) Bit beachtet werden.

3.1.4 High- und Low-Byte

Was passiert nun aber, wenn Datenwerte größer als 255 sind, sich also nicht in einem Byte unterbringen lassen? Wollen Sie z.B. bei einem Computer - der einen Speicherbereich von insgesamt 64 K hat - eine Adresse zwischenspeichern, so benötigen Sie einen Speicherplatz, der einen Zahlenwert bis zu 65535 aufnehmen kann. Zur Darstellung einer so großen Zahl sind dann zwei zusammenhängende Bytes - also insgesamt 16 Bits - notwendig.



Zu beachten ist, daß das 0. Bit des High-Byte dem 8. Bit der gesamten Zahl entspricht.

Da mit 16 Bits insgesamt 2↑16 Kombinationsmöglichkeiten für die beiden Ziffern 0 und 1 existieren, lassen sich also 2↑16 = 65536 verschiedene Zahlen ohne Berücksichtigung des Vorzeichens darstellen. Mit Vorzeichen ergibt sich ein Zahlenbereich von -32768 bis +32767, da das höchstwertige Bit (Bit Nr. 15) für das Vorzeichen verwendet wird.

3.2 Das Hexadezimal-System

Da die Darstellung größerer Zahlen im Binärsystem für den Menschen leicht sehr unübersichtlich wird und damit zu erhöhten Fehlermöglichkeiten führt, verwendet man in der Praxis der Programmierung andere Zahlensysteme. Besondere Bedeutung hat dabei das Hexadezimal-System.

Dies ist ein Zahlen-System, das auf 16 (hexadezimal = 16) verschiedenen Zahlzeichen aufgebaut ist. Wie im Dezimal-System gibt es die Ziffern 0 bis 9. Hinzu kommen die Buchstaben A bis F, die als Zahlzeichen den (dezimalen) Werten von 10 bis 15 entsprechen. Wieder bestimmt die Stelle innerhalb einer Zahl den Wert der Ziffer, nur muß man jetzt mit Potenzen zur Basis 16 rechnen:

```
A90F = 10 * 16↑3 + 9 * 16↑2 + 0 * 16↑1 + 15 * 16↑0
       (A)                                (F)

     = 43279 (dezimal).
```

In diesem System lassen sich also mit nur 4 Stellen recht große Zahlen darstellen. Für die Dezimalzahlen von 0 bis 255, für die man ja im Binär-System bis zu 8 Stellen braucht, reichen hier zwei Stellen aus: 255 (dezimal) = FF (hexadezimal). Die größte mit 4 Stellen darstellbare Zahl im Hexadezimal-System (also: FFFF) ist übrigens 65535 - das entspricht 64K, der gesamten Speicherkapazität eines üblichen 8-Bit-Computers, d.h. daß man mit einer 4-stelligen Hexadezimal-Zahl sämtliche Adressen (Speicherplätze) des Computers benennen kann.

Die Umrechnung einer Binärzahl in eine Hexadezimalzahl kann man leicht im Kopf durchführen: man teilt die Binärzahl (von rechts beginnend) in Vierergruppen, die also jeweils einen Wert zwischen 0 und 15 (dezimal) bzw. 0 und F (hexadezimal) darstellen und ordnet jeder Gruppe das entsprechende Hexadezimalzeichen zu.

Die folgende Tabelle enthält die Zahlen von 0 bis 32 in dezimaler, binärer und hexadezimaler Darstellung:

Dezimal	Binär	Hexadezimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1A
27	11011	1B
28	11100	1C
29	11101	1D
30	11110	1E
31	11111	1F
32	100000	20

4 | Wie schreibt man ein Programm in Assembler?

4. Wie schreibt man ein Programm in Assembler?

Betrachten wir als einfaches Beispiel die Addition zweier Zahlen A und B. In einer höheren Programmiersprache wie z.B. BASIC kann eine solche Berechnung z.B. so ausgeführt werden:

```
10 LET SUM = A + B
```

Anschließend enthält die Variable SUM als Ergebnis der Addition die Summe der Werte von A und B. In BASIC ist dazu zwar nur eine einzige Anweisung notwendig, jedoch kann man sich leicht klarmachen, daß der Rechner tatsächlich eine ganze Reihe von einzelnen Schritten ausführen muß, die man etwa folgendermaßen formulieren kann:

1. Hole den Wert aus Speicher A und bringe ihn in den Rechenspeicher.

2. Hole den Wert aus Speicher B und addiere ihn zum Inhalt des Rechenspeichers.

3. Hole den Wert aus dem Rechenspeicher und bringe ihn in den Speicher SUM.

4.1 Addition zweier Zahlen

Vielleicht fragen Sie sich jetzt, wozu diese Umstände notwendig sind. Dies ist darin begründet, daß die meisten 8-Bit-Prozessoren (dies gilt auch für die 6500-Prozessor-Familie) nicht gleichzeitig auf mehrere Speicheradressen zugreifen können.

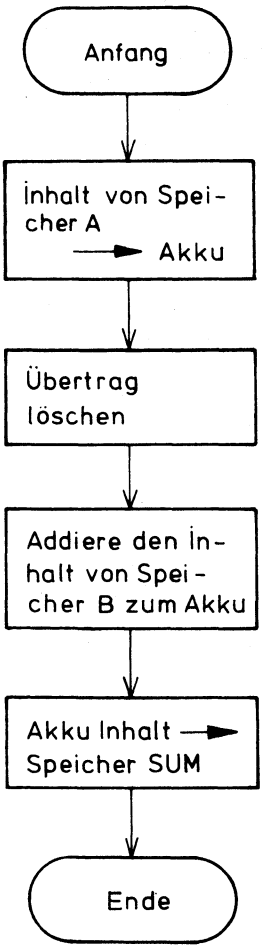
Wollen wir uns nun auf die Maschinenebene "hinab" begeben, so müssen wir für unseren Computer diese Anweisungen in ähnlicher Form formulieren:

1. Hole den Wert aus Speicher A und bringe ihn in den Akkumulator (= CPU-Rechenregister).

2. Bereite die CPU für die Addition vor (Übertrag löschen).

3. Addiere den Wert aus Speicher B zum Inhalt des Akkumulators und speichere das Ergebnis im Akkumulator.

4. Bringe den Wert aus dem Akkumulator in den Speicher SUM.



Übersetzt in die Maschinensprache sieht dieses Programm für unseren Computer letztlich so aus:

```
10100101
00010000
00011000
01100101
00010001
10000101
00010010
00000000
```

Bevor Sie nach dieser kleinen Kostprobe lieber gleich auf das Erlernen des Programmierens in Maschinensprache verzichten wollen, können wir Sie beruhigen: in dieser Form müssen Sie Ihrem Computer Anweisungen und Daten nämlich doch nicht servieren!

Als erstes stellen wir diese binären Informationen in einer anderen Form, nämlich hexadezimal, dar. Damit sieht die ganze Sache schon etwas "freundlicher" aus:

```
A5
10
18
65
11
85
12
00
```

Diese Darstellung ist zwar um einiges übersichtlicher, aber trotzdem ist es doch etwas schwierig für uns Nicht-Computer, festzustellen, daß hier eine Addition durchgeführt werden soll! Aber auch wenn wir die Bedeutung der einzelnen Hexadezimal-Zeichen kennen, wäre ein längeres Programm in dieser Schreibweise nicht mehr zu überblicken. Deshalb verwendet man für die einzelnen Anweisungen Schlüsselwörter, die im allgemeinen aus bis zu drei Buchstaben bestehen. Diese Schlüsselwörter - auch Mnemonics genannt - bieten eine viel bessere Gedankenstütze als die Hexadezimalzahlen, da die Bezeichnungen jeweils Abkürzungen für die betreffenden Aktionen darstellen, wie z.B. LDA für LoaD (= lade) Akkumulator.

LDA ist auch die erste Anweisung in unserem Beispielprogramm, wie Sie der nun folgenden Programm-Version entnehmen können, die den beiden bisher bekannten Fassungen gegenübergestellt ist (das Semikolon leitet einen Kommentar ein):

```
                    ;ADDITION ZWEIER ZAHLEN
10100101  A5 10       LDA $10
00010000
00011000  18          CLC
01100101  65 11       ADC $11
00010001
10000101  85 12       STA $12
00010010
00000000  00          BRK
```

Schauen wir uns an, was diese Anweisungen im einzelnen bewirken!

LDA

LDA $10 lädt den Wert aus dem Speicher mit der Adresse $10 in den Akkumulator. Das Dollarzeichen besagt, daß die angegebene Zahl als Hexadezimalzahl zu interpretieren ist.

CLC

CLC ist die Abkürzung für CLear Carry (= Übertrag löschen). Diese Anweisung ist für einfache 8-Bit-Additionen notwendig, da der 6510-Prozessor nur einen Additionsbefehl kennt, der einen eventuellen Übertrag (der sich als sogenanntes C-Flag im Prozessor-Status-Register befindet) aus vorherigen Aktionen zum Ergebnis hinzu addiert.

ADC

ADC $11 steht für ADdition with Carry (= Addition mit Übertrag) und bewirkt, daß der Inhalt aus dem Speicher $11 zum Inhalt des Akkumulators addiert wird. Dazu wird dann noch das Carry-Bit (0 oder 1) addiert. Das Ergebnis steht im Akkumulator. Hätten wir in unserem Fall das Carry-Bit vorher nicht gelöscht, könnte ein falsches Ergebnis herauskommen, nämlich dann, wenn das Carry-Bit aus einer vorhergehenden Aktion gesetzt war, also den Wert 1 hatte. Nach der Addition wird das Carry-Bit aktualisiert.

STA

STA $12 ist die Abkürzung für STore Accumulator (= speichere den Inhalt des Akkumulators). Dadurch wird der Inhalt des Akkumulators unter der angegebenen Adresse gespeichert.

BRK

BRK bedeutet BReaK, also Unterbrechung der Programmausführung durch Software. Dieser Befehl bewirkt außerdem, daß das B-Flag (Bit 4) des Prozessor-Status-Registers (P) gleich 1 gesetzt wird. Damit ist es möglich festzustellen, ob eine Unterbrechung (Interrupt) software- oder hardwaremäßig ausgelöst wurde (vgl. Kap. 9).

Um das Programm ohne Assembler einzugeben, können Sie das folgende BASIC-Ladeprogramm verwenden, mit dem die einzelnen Befehle in dezimaler Form mit POKE in den Speicher gebracht werden. Als Anfangsadresse wurde $C000 (49152 dezimal) gewählt, da hier ein 4 K großer RAM-Bereich beginnt, der von BASIC nicht benutzt wird. Dieser Speicherbereich reicht bis $CFFF (53247 dezimal) und eignet sich also auch für sehr umfangreiche Programme in Maschinensprache.

```
1 REM ..... ADDITION ZWEIER ZAHLEN
10 DATA 165, 16, 24,101, 17,133, 18,  0
20 FOR I = 49152 TO 49159
30 READ K: POKE I,K
40 SUM=SUM+K
50 NEXT I
60 IF SUM <> 474 THEN PRINT "PRUEFSUMME FALSCH!":STOP
70 PRINT "PRUEFSUMME RICHTIG!"
```

Der Aufruf des Programmes erfolgt von BASIC aus mit SYS 49159. Zum Testen können Sie vorher mit POKE jeweils einen Zahlenwert in die Adressen $10 (dezimal 16) und $11 (dezimal 17) bringen. Die Summe können Sie dann nach dem Programmaufruf mit PRINT PEEK(18) - also $12 - erhalten. Wenn das Ergebnis richtig sein soll, müssen Sie darauf achten, daß die Summe (vorläufig!) höchstens 255 betragen darf.

Eine direkte Eingabe des Hexadezimal-Codes ist mit dem im Anhang abgedruckten Programm möglich. Mit diesem Programm können Sie auch Ihre eigenen oder andere im Speicher stehenden Programme disassemblieren, d.h. in Assemblerschreibweise umsetzen lassen.

4.2 Addition mehrerer Zahlen

Nachdem wir nun wissen, wie man zwei Zahlen addiert, wollen wir uns überlegen, wie ein Programm aussehen könnte, mit dem man mehrere Zahlen addieren kann.

Zunächst werden wir versuchen, ein solches Programm nur mit unseren bisherigen Kenntnissen aufzustellen. Dies ist zwar grundsätzlich möglich, praktisch jedoch nur dann durchführbar, wenn es sich lediglich um wenige Zahlen handelt.

Deshalb werden wir anschließend zeigen, wie sich diese Aufgabe mit Hilfe zusätzlicher Befehle sehr viel einfacher und effektiver lösen läßt.

Welche Informationen brauchen wir, um mehrere Zahlen zu addieren?

Zunächst einmal müssen wir natürlich wissen, wo die zu addierenden Zahlen stehen. Um die Sache zu vereinfachen, gehen wir davon aus, daß die Daten - sinnvollerweise - zusammenhängend gespeichert sind. In dem Fall müssen wir nur die Anfangsadresse kennen und wissen, um wieviele Daten es sich handelt und wohin das Ergebnis gespeichert werden soll.

Folgende Vereinbarungen sollen für unser Programm gelten: das Programm soll bei der Speicheradresse $C000 beginnen, die Daten ab Adresse $C803 abgelegt sein, die Anzahl der Daten in Adresse $C800 und das Ergebnis in $C801 stehen.

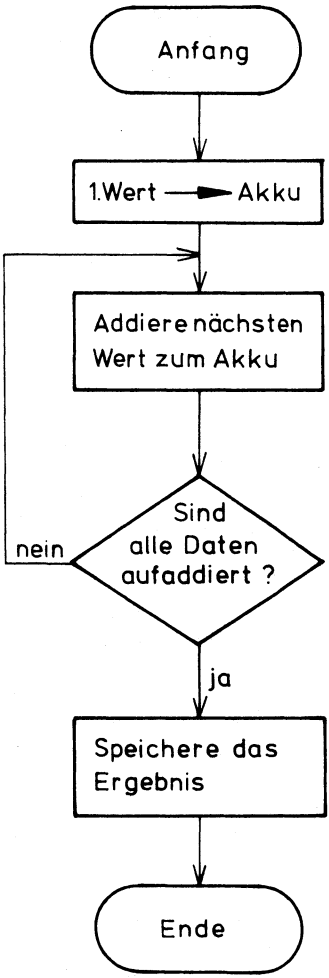
Damit kann der Algorithmus zur Addition angegeben werden:

1. Hole den Wert aus dem ersten Datenspeicher (Adresse $C803) und bringe ihn in den Akkumulator.

2. Addiere den nächsten Wert zum Inhalt des Akkumulators.

3. Wiederhole Schritt 2 solange, bis alle Daten zum Akkumulatorinhalt addiert sind.

4. Speichere den Inhalt des Akkumulators in Adresse $C801.


Anfang
1.Wert
Akku
Addiere nächsten Wert zum Akku
Sind alle Daten aufaddiert ?
nein
ja
Speichere das Ergebnis
Ende

Wollten wir diese mehrfache Addition mit unseren bisherigen Programmierkenntnissen lösen, so könnte das Programm etwa so aussehen:

```
C000   18            CLC          ;Addition vorbereiten
C001   AD 03 C8      LDA $C803    ;1.Summand (Low-Byte)
C004   6D 04 C8      ADC $C804    ;1.Summand (High-Byte)
C007   6D 05 C8      ADC $C805    ;2.Summand (Low-Byte)
C00A   6D 06 C8      ADC $C806    ;2.Summand (High-Byte)
C00D   ...
 ...   ...
 ...   ...
 ...   ...
 ...   ...
 ...
 ...   8D 01 C8      STA $C801    ;Ergebnisspeicher
 ...   00            BRK
```

Wenn Sie dieses Programms mit unserem ersten einfachen Additions-Programm vergleichen, werden Sie feststellen, daß es sich in den Hex-Codes von diesem an einigen Stellen unterscheidet, obwohl die mnemotechnischen Bezeichnungen übereinstimmen. Der Grund dafür ist in einer Besonderheit der 6500-Prozessoren zu sehen: die ersten 256 Speicheradressen ($00 - $FF), d.h. die sogenannte Null-Seite oder Zero Page, werden besonders behandelt. Alle Operationen innerhalb der Zero Page benötigen für die Angabe der Adresse nur ein Byte. Deshalb sehen die Befehls-Codes für die Zero Page anders aus als für andere Speicherbereiche. So heißt z.B. die Hexadezimal-Darstellung von LDA auf der Zero Page A5, ansonsten jedoch AD. Durch diese Möglichkeiten der Zero Page - Adressierung werden weniger Speicherplätze für das Programm benötigt und außerdem die Ausführungszeiten der Befehle in den meisten Fällen kürzer.

Dagegen sind für alle Adressen außerhalb der Zero Page zwei Bytes (also 16 Bit) erforderlich. Dabei wird das niederwertige oder Low-Order-Byte in dem Speicher mit der niedrigeren Adresse abgelegt, so daß der Hexcode für den Befehl LDA $C803 wie folgt aussieht:

```
AD 03 C8
```

Wie Sie sich leicht selbst überzeugen können, gefällt es Ihnen aber sicherlich nicht, in der soeben beschriebenen Weise z.B. 100 Zahlen zu addieren, da Sie für jede einzelne Zahl immer wieder die gleichen Befehle eingeben müßten. Abgesehen von Ihrer Geduld ist bei diesem Vorgehen auch die Speicherkapazität des Computers irgendwann erschöpft. Sehen wir also zu, daß wir einen eleganteren Lösungsweg finden!

Bisher haben wir jeden Datenspeicher durch die direkte Angabe der Adresse (LDA $C803, LDA $C804, ...) angesprochen. Diese Form der Adressierung nennt man direkte Adressierung (direct addressing), da die Adresse direkt angegeben wird.

In unserem Beispiel wäre es jedoch vorteilhafter, wenn wir nur die Anfangsadresse angeben könnten und mit Hilfe eines Zählers auf den jeweils folgenden Datenwert zugreifen könnten. Dieses Verfahren wird auch als indizierte Adressierung (indexed addressing) bezeichnet. Für diese Adressierungsart werden die X- und Y-Index-Register benötigt. Die eigentliche Adresse erhält man bei der indizierten Adressierung dadurch, daß der Inhalt des betreffenden Index-Registers zu der angegebenen Adresse addiert wird. Vor dem Datenzugriff kann man den Inhalt des Indexregisters verändern und hat dadurch einen Zugriff auf andere Speicher.

```
                    ;ADDITION MEHRERER ZAHLEN (SUMME (= 255)
C000   A2 00          LDX #$00     ;Zeiger auf Daten
C002   8A              TXA
C003   18          LOOP CLC        ;Addition vorbereiten
C004   7D 03 C8        ADC $C803,X ;Addition
C007   E8              INX         ;nächstes Byte
C008   EC 00 C8        CPX $C800   ;Zähler=Datenzahl?
C00B   D0  F6          BNE LOOP    ;nein: weiter
C00D   8D 01 C8        STA $C801   ;ja: Ergebnis!
C010   00              BRK
```

Es folgt eine kurze Beschreibung der neuen Befehle:

LDX

LDX (LoaD index register X with memory) ist - ähnlich wie LDA - ein Ladebefehl, der dazu dient, den Inhalt des Index-Registers X mit dem angegebenen Wert zu laden. Das Zeichen # zeigt an, daß der Wert $00 unmittelbar gemeint ist.

TXA (Transfer index register X to Accumulator) kopiert den Inhalt des Index-Registers X in den Akkumulator.

TXA

INX (INcrement index register X) erhöht den Inhalt des Index-Registers X um 1. Ist der Inhalt vor der Erhöhung bereits gleich 255 (= $FF hexadezimal), so wird er durch die Erhöhung auf Null und das Zero-Flag auf 1 gesetzt. Falls das 7. Bit gesetzt ist, wird das N-Flag (Vorzeichen-Flag) gesetzt.

INX

CPX (ComPare index register X with memory) vergleicht den Inhalt des Index-Registers X mit dem Inhalt des Speichers. Sämtliche Flags des Prozessor-Statuswortes werden in Abhängigkeit vom Vergleichsergebnis aktualisiert.

CPX

BNE (Branch on Not Equal to zero) bewirkt eine Verzweigung zu der angegebenen Sprungstelle (relativer Sprung), wenn das Zero-Flag nicht gesetzt ist. Das Programm wird an der Adresse fortgesetzt, die sich durch Addition des nach BNE angegebenen Abstandes zur Adresse des folgenden Befehls ergibt. Da der Abstand im Zweierkomplement (-128 bis +127) angegeben wird, ist eine Verzweigung im Bereich von -126 bis + 129 Bytes um den BNE-Befehl herum möglich.

BNE

Nach dem BNE-Befehl ist im Assembler-Programm die symbolische Adresse LOOP (Schleife) angegeben, die sich auf eine an anderer Stelle im Programm stehende Markierung bezieht. Die englische Bezeichnung für eine solche "Marke" heißt "Label". Während der Assemblierung (also der Übersetzung des Assemblerprogrammes in die Maschinensprache) ersetzt der Assembler jedes Label durch die entsprechende Speicheradresse. Diese Möglichkeit eines Assemblers erleichtert die Programmierung natürlich ganz erheblich.

Mit dem folgenden BASIC-Ladeprogramm können Sie das Programm zur Addition mehrerer Zahlen eingeben:

```
1 REM ......... ADDITION MEHRERER ZAHLEN (SUMME <= 255)
10 DATA 162,  0,138, 24,125,  3,200,232,236,  0
20 DATA 200,208,246,141,  1,200,  0
30 FOR I = 49152 TO 49168
40 READ K: POKE I,K
50 SUM=SUM+K
60 NEXT I
70 IF SUM <> 2116 THEN PRINT "PRUEFSUMME FALSCH!":END
80 PRINT "PRUEFSUMME RICHTIG!"
```

Der Aufruf von BASIC aus erfolgt wieder mit SYS 49152. Zum Testen können Sie die zu addierenden Zahlen mit POKE in die Speicherplätze ab 51203 ($C803) bringen. Die Anzahl der Summanden muß in 51200 stehen. Beachten Sie, daß die Summe höchstens 255 betragen darf. Das Ergebnis steht nach dem Programmaufruf wieder in 51201 und 51202.

Mit diesem Programm können Sie bis zu 256 Zahlen addieren, wobei allerdings zu beachten ist, daß die Summe nicht größer als 255 (!) sein darf. Sollten Sie sich dadurch allzusehr eingeschränkt fühlen, so können Sie im nächsten Abschnitt erfahren, wie Sie mit größeren Zahlen umgehen können!

Wird das Ergebnis einer Addition größer als 255, so reicht ein Byte nicht mehr aus, um eine solche Zahl darzustellen. Mit zwei Bytes, also 16 Bits, kann man Zahlen bis maximal 65535 darstellen. Sofern die einzelnen Summanden nicht größer sind als 255, müssen wir folgende Änderungen vornehmen: zunächst einmal muß das obige Programm um ein zusätzliches Byte für das Additionsergebnis ergänzt werden. Dazu verwenden wir den bisher freigehaltenen Speicherplatz $C802.

```
                ;ADDITION MEHRERER ZAHLEN (SUMME (= 65535)
C000   A2 00            LDX #$00     ;Löschen: X- ...
C002   8A               TXA          ;Akku- ...
C003   A8               TAY          ;Y-Register
C004   18        LOOP   CLC          ;Addition vorb.
C005   7D 03 C8         ADC $C803,X  ;Low-Byte addieren
C008   90 01            BCC WEITER   ;Verzweige, wenn
                                      kein Übertrag
C00A   C8               INY          ;Übertrag
C00B   E8        WEITER INX          ;nächstes Byte
C00C   EC 00 C8         CPX $C800    ;Zähler=Anzahl?
C00F   D0 F3            BNE LOOP     ;nein: LOOP
```

```
C011   8D 01 C8        STA $C801    ;Ergebnis: Low-Byte
C014   8C 02 C8        STY $C802    ;High-Byte
C017   00               BRK
```

TAY

Der erste neue Befehl ist unter der Adresse $C003 zu finden: TAY steht für Transfer Accumulator to index register Y, d.h. der Inhalt des Akkumulators wird ins Index-Register Y kopiert.

BCC

BCC bedeutet Branch Carry Clear und bewirkt in unserem Beispiel eine Verzweigung zu dem übernächsten Befehl, wenn das Carry-Bit gelöscht ist, also durch die Addition nicht gesetzt wurde. Das Programm wird an der Adresse fortgesetzt, die sich durch Addition des nach BCC angegebenen Abstandes zur Adresse des folgenden Befehls ergibt. Bei einem Rücksprung erfolgt die Angabe der zu überspringenden Speicherstellen im Zweierkomplement. Sollte z.B. ein Sprung um zwei Bytes zurück erfolgen, so müßte in unserem Fall in Speicheradresse $C009 statt des Wertes $01 der Wert $FE angegeben werden. ($FE ist gleich -2 im Zweierkomplement).

INY

INY (INcrement index register Y) erhöht den Inhalt des Index-Registers Y um 1. Ist der Inhalt vor der Erhöhung bereits gleich 255 (= $FF hexadezimal), so wird er durch die Erhöhung auf Null und das Zero-Flag auf 1 gesetzt. Falls das 7. Bit gesetzt ist, wird das N-Flag (Vorzeichen-Flag) gesetzt.

STY

STY ist die Abkürzung für STore index register Y (= speichere den Inhalt des Index-Registers Y). Der Inhalt des Y-Registers wird unter der angegebenen Adresse gespeichert.

In diesem Programm haben wir zum ersten Mal das zweite Index-Register, das Y-Register, benutzt. Dieses Register wird hier zur Zwischenspeicherung für das High-Byte des Ergebnisses verwendet. Dazu muß das Y-Register vorher gelöscht werden (durch TAY, da wir wissen, daß der Akkumulator den Wert 0 enthält. Selbstverständlich könnte der Wert 0 auch explizit ins Y-Register geladen werden, was jedoch ein Byte zusätzlich erfordern würde.)

Nach jeder Addition wird der Zustand des Carry-Bits danach abgefragt, ob ein Übertrag entstanden ist. Wenn das der Fall ist, wird der Inhalt des Index-Registers Y (durch den Befehl INY) um 1 erhöht, da jeder Übertrag dem Wert 256 entspricht. Ansonsten wird der Befehl INY übersprungen und das Y-Register nicht verändert. Nach dem Durchlauf der Schleife müssen wir dafür sorgen, daß der Inhalt dieses Registers im Ergebnis berücksichtigt wird. Dazu wird dieser Wert, der das höherwertige Byte des Additions-Ergebnisses darstellt, in den dafür vorgesehenen Speicher $C002 geschrieben.

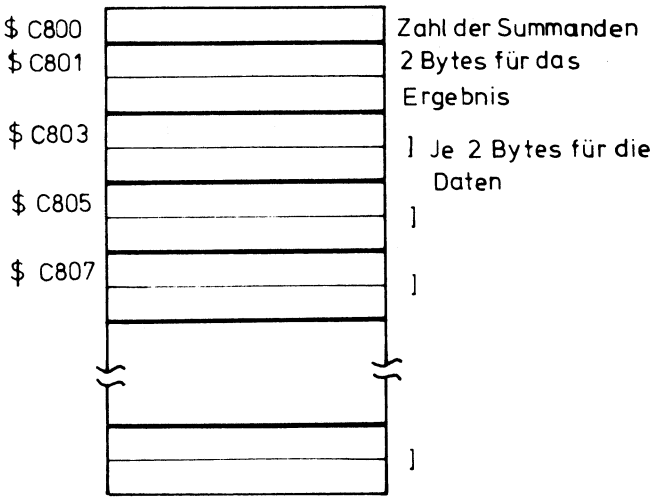
Mit dem folgenden BASIC-Ladeprogramm können Sie das Programm zur Addition mehrerer Zahlen in den Speicher bringen:

```
1 REM ..... ADDITION MEHRERER ZAHLEN (SUMME <= 65535)
10 DATA 162,  0,138,168, 24,125,  3,200,144,  1
20 DATA 200,232,236,  0,200,208,243,141,  1,200
30 DATA 140,  2,200,  0
40 FOR I = 49152 TO 49175
50 READ K: POKE I,K
60 SUM=SUM+K
70 NEXT I
80 IF SUM <> 2968 THEN PRINT "PRUEFSUMME FALSCH!":END
90 PRINT "PRUEFSUMME RICHTIG!"
```

4.3 16-Bit Addition

Nachdem nun die Summe der zu addierenden Zahlen größer sein darf als 255, soll das Programm jetzt so geändert werden, daß auch die einzelnen Summanden größer sein dürfen als 255. Dies ist einfach dadurch möglich, daß für jeden Summanden zwei Bytes reserviert werden. In unserem Beispiel bedeutet das, daß der erste Summand die beiden Speicherplätze $C803 und $C804, der zweite die Plätze $C805 und $C806 belegt usw.

Wie bisher muß die Zahl der Summanden in Adresse $C800 stehen. Für das Ergebnis werden jetzt die beiden Speicherplätze $C801 und $C802 vorgesehen. Ab $C803 befinden sich die Daten (Summanden), die jeweils zwei Bytes belegen. Diese Speicheraufteilung ist in der folgenden Abbildung dargestellt:



Als erstes muß das Index-Register X gelöscht, d.h. auf Null gesetzt werden, da dieses Register als "Zeiger" für die einzulesenden Daten benutzt wird. Außerdem werden die beiden für das Ergebnis vorgesehenen Speicher $C801 und $C802 dadurch gelöscht, daß der augenblickliche Wert des X-Registers (Null) mit dem Befehl STX (STore index X in memory) in diese Speicher geschrieben wird:

STX

```
C000   A2 00        LDX #$00     ;X-Register und ...
C002   8E 01 C8     STX $C801    ;Ergebnisspeicher ...
C005   8E 02 C8     STX $C802    ;löschen
```

Da jeder Summand nun aber zwei Bytes belegt, d.h. die Anzahl der Summanden nicht der Anzahl der benutzten Speicherplätze entspricht, müssen wir den Inhalt von $C800 verdoppeln. Zunächst muß der Prozessor mit CLC für eine einfache Addition vorbereitet werden. Daraufhin wird der Akkumulator mit der Anzahl der Summanden geladen, die in Adresse $C800 stehen. Dieser Wert wird dann zum Inhalt des Akkumulators addiert und das Ergebnis wieder in $C800 gespeichert:

```
C008   18            CLC          ;Addition vorbereiten
C009   AD 00 C8      LDA $C800    ;Bytezähler ...
C00C   6D 00 C8      ADC $C800    ;verdoppeln
C00F   8D 00 C8      STA $C800    ;zurückspeichern
```

Innnerhalb der nun folgenden (durch LOOP markierten) Programm-Schleife erfolgt die eigentliche Berechnung der Summe.

Als erstes muß immer wieder das Carry-Flag gelöscht werden, da bei der Addition der Low-Bytes sonst ein Übertrag aus vorherigen Berechnungen zu Fehlern führen könnte.

Nach der Addition des Low-Bytes des jeweiligen Summanden zum Speicherplatz für das Low-Byte-Ergebnis wird der Zeiger (X-Register) um eins erhöht und zeigt somit auf das High-Byte.

Auf ähnliche Weise wird das High-Byte des Summanden zum Speicherplatz für das Ergebnis des High-Bytes hinzuaddiert, wobei ein eventueller Übertrag aus der Low-Byte-Addition berücksichtigt wird. Anschließend wird wieder der Inhalt des X-Registers um eins erhöht und zeigt nun auf das Low-Byte des nächsten Summanden.

Dieser Vorgang wiederholt sich solange, bis der Zeiger für die Summanden (Index-Register X) der Byte-Anzahl der Summanden entspricht. Diese Verzweigung wird durch die Befehle CPX und BNE gesteuert. Durch den Vergleichsbefehl CPX wird unter anderem das Zero-Flag des Status-Registers aktualisiert. Der Zustand des Zero-Flags wird dann durch den Befehl BNE abgefragt. Eine Verzweigung findet statt, wenn das Zero-Flag nicht gesetzt ist:

```
C012   18          LOOP CLC          ;Addition vorbereiten
C013   AD 01 C8       LDA $C801    ;Low-Byte ...
C016   7D 03 C8       ADC $C803,X  ;addieren
C019   8D 01 C8       STA $C801    ;zurückspeichern
C01C   E8             INX          ;nächstes Byte:
C01D   AD 02 C8       LDA $C802    ;High-Byte ...
C020   7D 03 C8       ADC $C803,X  ;addieren
C023   8D 02 C8       STA $C802    ;zurückspeichern
C026   E8             INX          ;Zähler erhöhen:
C027   EC 00 C8       CPX $C800    ;Endwert erreicht?
C02A   D0 E6          BNE LOOP     ;nein: LOOP
```

Im folgenden ist noch einmal das ganze Programm wiedergegeben:

```
                ;16-BIT-ADDITION
C000   A2 00            LDX #$00     ;X-Register und ...
C002   8E 01 C8         STX $C801    ;Ergebnisspeicher ...
C005   8E 02 C8         STX $C802    ;löschen
C008   18               CLC          ;Addition vorbereiten
C009   AD 00 C8         LDA $C800    ;Bytezähler ...
C00C   6D 00 C8         ADC $C800    ;verdoppeln
C00F   8D 00 C8         STA $C800    ;zurückspeichern
C012   18          LOOP CLC          ;Addition vorbereiten
C013   AD 01 C8         LDA $C801    ;Low-Byte ...
C016   7D 03 C8         ADC $C803,X  ;addieren
C019   8D 01 C8         STA $C801    ;zurückspeichern
C01C   E8               INX          ;nächstes Byte:
C01D   AD 02 C8         LDA $C802    ;High-Byte ...
C020   7D 03 C8         ADC $C803,X  ;addieren
C023   8D 02 C8         STA $C802    ;zurückspeichern
C026   E8               INX          ;Zähler erhöhen:
C027   EC 00 C8         CPX $C800    ;Endwert erreicht?
C02A   D0 E6            BNE LOOP     ;nein: LOOP
C02C   00               BRK
```

Hier das dazugehörige BASIC-Ladeprogramm:

```
1 REM ..... 16-BIT-ADDITION
10 DATA 162,  0,142,  1,200,142,  2,200, 24,173
20 DATA   0,200,109,  0,200,141,  0,200, 24,173
30 DATA   1,200,125,  3,200,141,  1,200,232,173
40 DATA   2,200,125,  3,200,141,  2,200,232,236
50 DATA   0,200,208,230,  0
60 FOR I = 49152 TO 49196
70 READ K: POKE I,K
80 SUM=SUM+K
90 NEXT I
100 IF SUM<> 5348 THEN PRINT "PRUEFSUMME FALSCH!":END
110 PRINT "PRUEFSUMME RICHTIG!"
```

Zum Testen können Sie 16-Bit-Zahlen in die Speicherplätze ab Adresse 51203 schreiben. Dabei ist zu beachten, daß das Low-Byte einer Zahl jeweils vor dem High-Byte steht. Nach dem Programmaufruf steht das Ergebnis in den beiden Speicherplätzen 51001 (Low-Byte) und 51002 (High-Byte).

Mit diesem Programm können wir jetzt zwar größere Zahlen als bisher aufaddieren, wobei jedoch zu beachten ist, daß die Summe nicht größer als 65535 werden darf, d.h. 2↑16 - 1, weil dies die größte Zahl ist, die durch 16 Bits dargestellt werden kann.

Da das Index-Register X, das uns als Zeiger dient, nur aus acht Bit besteht, sind damit nur Zahlen bis zu 255 darstellbar (= 2↑8 - 1). Entsprechend ließen sich damit im vorletzten Beispiel bis zu 255 Zahlen addieren, die aus jeweils einem Byte bestanden. Da wir es hier nun mit Summanden aus zwei Bytes zu tun haben, lassen sich nur maximal 127 Summanden addieren. Wie wir auch diese Einschränkung noch überwinden können, erfahren Sie im übernächsten Abschnitt, nachdem wir uns zuvor mit dem modularen Aufbau von Programmen beschäftigt haben. Eine wichtige Technik dazu stellen die Unterprogramme dar, die deshalb zunächst besprochen werden sollen.

5 | Unterprogramme

5. Unterprogramme

Die bisherigen Programme besaßen alle wesentliche Einschränkungen, die ihre Anwendbarkeit auf einen jeweils nur ganz bestimmten Bereich ermöglichten. Um nun allgemeiner verwendbare Programme entwickeln zu können, werden wir im folgenden Programmiertechniken kennenlernen, die bei der Entwicklung von modular aufgebauten Programmen eine große Rolle spielen.

Wird in einem Programm eine bestimmte Folge von Befehlen mehrmals benötigt, so ist es offensichtlich nicht sinnvoll, diese identischen Befehlsfolgen immer wieder neu in das Programm einzufügen, da dies zu einem unnötigen Verbrauch von Speicherplatz im Arbeitsspeicher des Computers führt. Außerdem entsteht auf diese Weise unnötiger Schreibaufwand. Noch wichtiger aber ist die Tatsache, daß die Programme erheblich übersichtlicher werden, wenn eine solche sich wiederholende Anweisungsfolge nur einmal im Arbeitsspeicher steht und dann von jeder gewünschten Stelle des aufrufenden Programms aufgerufen werden kann: dieser sich wiederholende Programmteil wird allgemein als Unterprogramm (subroutine) bezeichnet.

Woran erkennt der Prozessor aber, wann ein Unterprogramm abgearbeitet werden soll und an welcher Stelle sich Anfang und Ende des Unterprogramms im Arbeitsspeicher befinden?

Die Ausführung eines Unterprogramms wird durch einen speziellen Befehl eingeleitet, der die Anfangsadresse des betreffenden Unterprogramms enthält: dieser Befehl heißt in unserem Fall JSR (Jump to SubRoutine), also Sprung zum Unterprogramm.

JSR

Wenn ein solcher Befehl im Programm ausgeführt wird, laufen in der Befehlsabarbeitung eine ganze Reihe von Aktionen ab:

Zuerst muß der Prozessor dafür sorgen, daß er weiß, an welcher Stelle im aufrufenden Programm die Programmabarbeitung nach der Ausführung des Unterprogramms fortgesetzt werden muß, d.h. diese Adresse muß zunächst einmal notiert werden. Dies geschieht ohne Zutun des Programmierers im Stack (Stapel-Speicher, vgl. Beschreibung des Prozessoraufbaus). Diese wichtige Datenstruktur wird gleich ausführlicher beschrieben.

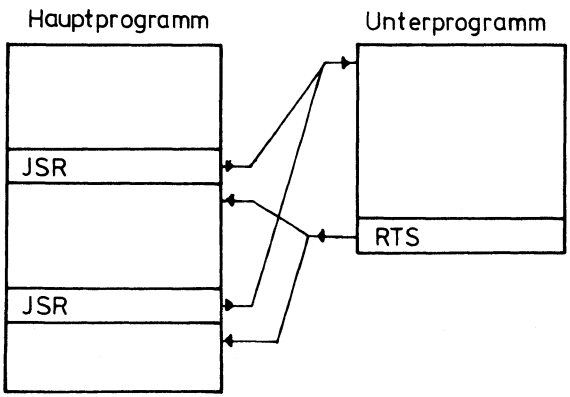
Dann wird die Anfangsadresse des Unterprogramms, die ja im JSR-Befehl angegeben ist, in den Programmzähler geladen, womit eine Verzweigung zu der angegebenen Unterprogramm-Adresse bewirkt wird.

Im Unterprogramm selbst wird das Ende des Unterprogramms durch den Befehl RTS (ReTurn from Subroutine) markiert, d.h. Rücksprung vom Unterprogramm.

RTS Dieser Befehl lädt den Programmzähler (PC) mit der Adresse im Stack, die durch den Aufruf dieses Unterprogramms gespeichert worden ist (s. JSR).

Auf diese Weise kann also ein Unterprogramm aus der Sicht des Programmierers wie ein einziger Befehl betrachtet werden, wodurch die Übersichtlichkeit des gesamten Programms deutlich erhöht wird.

Die folgende Abbildung soll die Vorteile der Unterprogramm-Technik noch einmal grafisch illustrieren:



Wie schon erwähnt, befindet sich der Speicherbereich für den Stack in den 256 Byte-Speichern mit den Adressen von $1FF bis $100. Nach dem Einschalten des Computers muß der Anfang dieses Stack-Bereiches definiert werden: dies geschieht dadurch, daß der Stack-Pointer auf die Anfangs-Adresse $1FF gesetzt wird, d.h. der Stack wird von oben nach unten benutzt. Die folgenden Befehle bewirken dieses Initialisieren des Stack-Pointers:

```
LDX #$FF
TXS
```

TXS

Der Befehl TXS (Transfer X into S) bedeutet: kopiere den Inhalt des X-Registers in den Stack-Pointer. Dies ist die einzige Möglichkeit, den Inhalt des Stack-Pointers direkt zu beeinflussen. Dieser Befehl ist unbedingt notwendig, um den Stack-Pointer in einen bestimmten Zustand zu bringen, z.B. nach dem Einschalten des Computers (Initialisierung). Natürlich muß man dem X-Register vorher einen entsprechenden Wert zuweisen, beim Initialisieren üblicherweise den Wert $FF.

TSX

Da der Stack-Bereich des 6510-Prozessors begrenzt ist (256 Bytes), muß man darauf achten, daß der Stack nicht "überläuft". Dazu ist es wichtig, den Zustand des Stack-Pointers abzufragen, wenn er in kritische Bereiche zeigt, um im Programm entsprechend reagieren zu können. Dies geschieht, indem man den Inhalt des Stack-Pointers ins X-Register kopiert und diesen Wert dann abfragt (da man keinen direkten Zugriff auf den Stack-Pointer hat). Die Übertragung des Stack-Pointers ins X-Register geschieht mit dem Befehl TSX (Transfer S into X) - kopiere den Inhalt des Stack-Pointers ins Y-Register. Dieser Befehl stellt das Gegenstück zu dem soeben vorgestellten Befehl TXS dar.

5.1 Parameterübergabe

Im allgemeinen soll ein Unterprogramm Daten aus dem aufrufenden Programm bearbeiten. Die Übergabe dieser Parameter kann auf verschiedene Weise erfolgen:

1. Parameterübergabe durch Register
2. Parameterübergabe durch den Stack
3. Parameterübergabe durch vereinbarte Speicher
4. direkte Parameterübergabe

Bei der Parameterübergabe durch Register hat man den Vorteil, daß das Unterprogramm unabhängig vom Speicher bleibt. Allerdings ist diese Möglichkeit beim 6510 durch die geringe Zahl der internen Register stark eingeschränkt.

Die Parameterübergabe durch den Stack hat denselben Vorteil wie die Übergabe durch Register, ist also ebenfalls speicherunabhängig. Nachteilig ist, daß die Anzahl der verschachtelten Unterprogramme reduziert wird.

Benutzt man vereinbarte Speicher, so kann die Zahl der Daten erheblich größer sein als bei den vorigen Techniken. Der Nachteil liegt darin, daß das Unterprogramm dann an diesen Speicherbereich gebunden ist.

Bei der direkten Parameterübergabe stehen die zu übergebenden Daten ebenfalls in festen Speicherstellen, jedoch direkt im aufrufenden Programm. Von dieser Möglichkeit wird in dem Programm ASSI in Kapitel 6.2 Gebrauch gemacht. Die ersten drei Möglichkeiten der Parameterübergabe hingegen werden im folgenden an kleinen Beispielprogrammen demonstriert.

Als Beispiel soll ein Unterprogramm zur Addition von zwei 16-Bit-Zahlen dienen. Dieses Unterprogramm soll die folgende Aufgabe erfüllen: vom aufrufenden Programm werden zwei 16-Bit-Zahlen übergeben, die im Unterprogramm addiert werden. Das Ergebnis wird dem aufrufenden Programm übergeben.

5.1.1 Parameterübergabe durch Register

Mit dem folgenden Programm wird über die Index-Register X und Y eine 16-Bit-Zahl an ein Unterprogramm übergeben, das diese Zahl zum Inhalt der Adressen $C801 und $C802 addiert. Zur Addition von zwei Zahlen muß das Unterprogramm also zweimal aufgerufen werden.

```
C000   A9 00        LDA #$00    ;Ergebnisspeicher ...
C002   8D 01 C8     STA $C801   ;löschen
C005   8D 02 C8     STA $C802
C008   AE 03 C8     LDX $C803   ;1.Summand (Low-Byte)
C00B   AC 04 C8     LDY $C804   ;1.Summand (High-Byte)
C00E   20 1B C0     JSR ADD     ;Addition
C011   AE 05 C8     LDX $C805   ;2.Summand (Low-Byte)
C014   AC 06 C8     LDY $C806   ;2.Summand (High-Byte)
C017   20 1B C0     JSR ADD     ;Addition
C01A   60           RTS
```

Zunächst werden die Speicherplätze $C801 und $C802, in denen das Ergebnis der Addition stehen soll, gelöscht. Die beiden zu addierenden Zahlen müssen unter den Adressen $C803 bis $C806 gespeichert sein. Nachdem das Low-Byte des ersten Summanden mit LDX ins X- und das High-Byte mit LDY (LoaD index Y with memory) ins Y-Register gebracht wurde, wird das Unterprogramm zur Addition aufgerufen. Anschließend wird der zweite Summand in die Index-Register gebracht und das Unterprogramm noch einmal aufgerufen. Die Summe steht danach in $C801 und $C802. Schließt man das aufrufende Hauptprogramm selbst mit dem Rücksprungbefehl RTS ab, so erfolgt anschließend der Rücksprung nach BASIC. Dies ist nützlich, wenn Sie Maschinenprogramme in BASIC-Programme einbinden wollen.

LDY

Das dazugehörige Unterprogramm kann wie folgt aussehen:

```
C01B   18           ADD  CLC         ;Addition vorbereiten
C01C   8A                TXA         ;Low-Byte zum ...
C01D   6D 01 C8          ADC $C801   ;Ergebnisspeicher add.
C020   8D 01 C8          STA $C801   ;und zurückspeichern
C023   98                TYA         ;High-Byte zum ...
C024   6D 02 C8          ADC $C802   ;Ergebnisspeicher add.
C027   8D 02 C8          STA $C802   ;und zurückspeichern
C02A   60                RTS
```

Vor der Addition des Low-Bytes wird das Carry-Flag gelöscht. Dann wird das Low-Byte mit TYA (Transfer index register Y to Accumulator, kopiere den Inhalt des Index-Registers Y in den Akkumulator) in den Akku gebracht und zum Inhalt des Ergebnisspeichers hinzuaddiert. Entsprechend wird mit dem High-Byte verfahren.

TYA

Mit dem folgenden BASIC-Ladeprogramm können Sie das Beispiel-Programm (einschließlich Unterprogramm) zur Parameterübergabe durch Register eingeben:

```
1 REM ..... PARAMETERUEBERGABE (REGISTER)
10 DATA 169,  0,141,  1,200 141,  2,200,174,  3
20 DATA 200,172,  4,200, 32, 27,192,174,  5,200
30 DATA 172,  6,200, 32, 27,192, 96, 24,138,109
40 DATA   1,200,141,  1,200,152,109,  2,200,141
50 DATA   2,200, 96
60 FOR I = 49152 TO 49194
70 READ K: POKE I,K
80 SUM=SUM+K
90 NEXT I
100 IF SUM <> 4678 THEN PRINT "PRUEFSUMME FALSCH!":END
110 PRINT "PRUEFSUMME RICHTIG!"
```

Das Programm wird von BASIC aus mit SYS 49152 aufgerufen. Zum Testen können Sie mit POKE-Anweisungen zwei 16-Bit-Zahlen in der Reihenfolge "Low-Byte, High-Byte" in die Speicher 51203 ($C803) bis 51206 ($C806) bringen.

Das Ergebnis können Sie nach dem Programmaufruf mit PRINT PEEK(51201)+PEEK(51202)*256 aus den Speicherstellen $C801 und $C802 erhalten.

5.1.2 Parameterübergabe durch den Stack

Bei dieser Art der Parameterübergabe wird der Datenaustausch zwischen dem aufrufenden und dem Unterprogramm dadurch realisiert, daß die Daten vor dem Unterprogrammaufruf in den Stack geladen werden. - Das nächste Programm illustriert diese Programmiertechnik.

So könnte in diesem Fall das aufrufende Programm aussehen:

```
C000   AD 03 C8      LDA $C803   ;Summanden ...
C003   48            PHA         ;in den Stack
C004   AD 04 C8      LDA $C804
C007   48            PHA
C008   AD 05 C8      LDA $C805
C00B   48            PHA
C00C   AD 06 C8      LDA $C806
C00F   48            PHA
C010   20 1C C0      JSR ADD     ;Addieren
C013   68            PLA         ;Ergebnis vom Stack...
C114   8D 02 C8      STA $C802   ;holen (High-Byte)
C017   68            PLA
C018   8D 01 C8      STA $C801   ;(Low-Byte)
C01B   60            RTS         ;Rückkehr ins auf-
                                 ;rufende (BASIC-)
                                 ;Programm
```

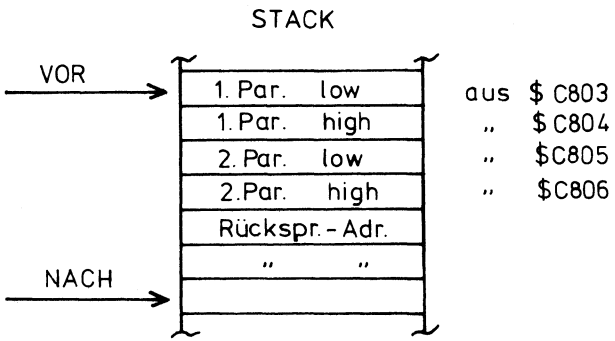
Die Operanden für die Addition werden vor dem Aufruf des Unterprogrammes in den Stack geschoben (Programmadresse $C000 - $C00F). Hierzu gibt es einen speziellen Befehl, nämlich PHA (PusH Accumulator on stack), der den Inhalt des Akkumulators in den Stack bringt. Durch diesen Befehl wird der Stack-Pointer geändert, und zwar um ein Byte erniedrigt. Bevor der Rücksprung zum aufrufenden Programm erfolgt, muß dafür gesorgt werden, daß der Stack-Pointer auf die richtige Speicherstelle zeigt, in der die Rücksprungadresse gespeichert ist. Anders gesagt: der gerettete Inhalt des Akkumulators muß wieder zurückgeholt werden. Dies geschieht durch den Befehl PLA (PuLl Accumulator from stack). Nach dem Laden des Akkumulators wird der Stack-Pointer um eins erhöht.

PHA

PLA

Vor der Ausführung des Unterprogrammes wird die Rücksprungadresse in den Stack gebracht, und zwar in der Form: Inhalt des aktuellen Programmzählers + 2 (oder anders gesagt: Rücksprungadresse -1). In unserem Fall ist dies also der Wert $C010 + $02.

Die folgende Abbildung zeigt die Anordnung der Parameter und Rücksprungadressen im Stack:



Das dazugehörige Unterprogramm kann wie folgt aussehen:

```
C01C   68          ADD  PLA          ;Rücksprungadresse ...
C01D   85 1A            STA $1A      ;retten
C01F   68               PLA
C020   85 1B            STA $1B
C022   68               PLA          ;2. High-Byte holen
C023   85 1C            STA $1C
C025   68               PLA          ;2. Low-Byte holen
C026   85 1D            STA $1D
C028   68               PLA          ;1. High-Byte holen
C029   85 1E            STA $1E
C02B   68               PLA          ;1. Low-Byte holen
C02C   18               CLC          ;Addition vorbereiten
C02D   65 1D            ADC $1D      ;Low-Bytes addieren
C02F   48               PHA          ;Ergebnis in den Stack
C030   A5 1C            LDA $1C      ;High-Bytes ...
C032   65 1E            ADC $1E      ;addieren
C034   48               PHA          ;Ergebnis in den Stack
C035   A5 1B            LDA $1B      ;Rücksprungadresse ...
C037   48               PHA          ;in den Stack ...
C038   A5 1A            LDA $1A      ;zurück-
C03A   48               PHA          ;bringen
C03B   60               RTS          ;Rücksprung zum auf-
                                     ;rufenden Programm
```

Zunächst wird die Rücksprungadresse zum aufrufenden Programm "gerettet", indem sie in die Speicher $1A und $1B gebracht wird.

Nun können die einzelnen Daten vom Stack geholt werden - auf Grund der Stack-Struktur allerdings in umgekehrter Reihenfolge! Während vom zweiten Summanden beide Bytes zwischengespeichert werden, genügt es, nur das High-Byte des ersten Summanden zu speichern, da das Low-Byte gleich zum Low-Byte des zweiten Summanden addiert werden kann. Das Low-Byte des Ergebnisses kann vom Akku aus gleich in den Stack zurückgebracht werden. Anschließend werden die beiden High-Bytes addiert, und die Summe ebenfalls in den Stack gebracht. Auf diese Weise gelangt das aufrufende Programm natürlich zuerst an das High-Byte des Ergebnisses, was im aufrufenden Programm berücksichtigt werden muß. Selbstverständlich hätte man durch Zwischenspeichern auch die gewohnte Reihenfolge "Low-Byte, High-Byte" einhalten können.

Vor dem Rücksprung zum aufrufenden Programm muß die in $1A und $1B gespeicherte Rücksprungadresse in den Stack zurückgeschrieben werden - und zwar in umgekehrter Reihenfolge wie beim Lesen aus dem Stack.

Hier wieder das entsprechende BASIC-Ladeprogramm:

```
1 REM ..... PARAMETERUEBERGABE (STACK)
10 DATA 173,  3,200, 72,173,  4,200, 72,173,  5
20 DATA 200,173,  6,200, 72, 32, 26,192,104,141
30 DATA   2,200,104,141,  1,200,104,133, 26,104
40 DATA 133, 27,104,133, 28,104,133, 29,104,133
50 DATA  30,104, 24,101, 29, 72,165, 28,101, 30
60 DATA  72,165, 26, 72,165, 27, 72, 96
70 FOR I = 49152 TO 49209
80 READ K: POKE I,K
90 SUM=SUM+K
100 NEXT I
110 IF SUM <>5543 THEN PRINT "PRUEFSUMME FALSCH!":END
120 PRINT "PRUEFSUMME RICHTIG!"
```

Der Aufruf des Programmes von BASIC aus erfolgt wieder mit SYS 49152. Zum Testen können Sie wie beim vorigen Beispiel zwei Summanden in den Speicher bringen und das Ergebnis aus 51201 ($C801) und 51202 ($C802) entnehmen.

5.1.3 Parameterübergabe durch vereinbarte Speicher

Im einfachsten Fall geschieht dies dadurch, daß die Parameter im Speicher mit festen Adressen abgelegt werden: nehmen wir an, daß die beiden zu addierenden Zahlen - d.h. die Eingangs-Parameter - in den Speicherplätzen $1A bis $1D gespeichert werden, und für das Ergebnis - die Ausgangs-Parameter - die Speicher $1E und $1F reserviert sind.

Vor dem Aufruf müsen die aktuellen Parameter in die vorgesehenen Speicherplätze $1A bis $1D gebracht werden, und nach dem Abarbeiten des Unterprogramms muß das Ergebnis (also in unserem Beispiel die Summe der beiden Zahlen) aus den Speicherplätzen $1E und $1F (dezimal: 20 und 21) geholt werden.

Das aufrufende Programm könnte im einfachsten Fall etwa so aussehen (wobei das Unterprogramm bei Adresse $C022 beginnen soll):

```
C000   AD 03 C8      LDA $C803   ;Summanden in die ...
C003   85 1A         STA $1A     ;vereinbarten ...
C005   AD 04 C8      LDA $C804   ;Speicher bringen
C008   85 1B         STA $1B
C00A   AD 05 C8      LDA $C805
C00D   85 1C         STA $1C
C00F   AD 06 C8      LDA $C806
C012   85 1D         STA $1D
C014   20 22 C0      JSR ADD     ;Addition
C017   A5 1E         LDA $1E     ;Ergebnis ...
C019   8D 01 C8      STA $C801   ;umspeichern
C01C   A5 1F         LDA $1F
C01E   8D 02 C8      STA $C802
C021   60            RTS
```

Díe ersten acht Befehle laden die entsprechenden Zahlen in die für die Eingangs-Parameter vorgesehenen Speicherplätze. Darauf folgt - mit dem Befehl JSR ADD - der Aufruf des Unterprogrammes, das bei Adresse $C022 beginnt. Anschließend wird der Ausgangs-Parameter, also das Ergebnis des Unterprogramms, in die gewünschten Speicherstellen gebracht.

Der aufmerksame Leser wird sich daran erinnern, warum bei manchen LDA- und STA-Befehlen statt drei nur zwei Bytes benötigt werden: hier handelt es sich um Befehle mit der Zero-Page-Adressierung. Durch die Verwendung dieser Befehle wird nicht nur Speicherplatz gespart, sondern auch die Ausführungszeit beschleunigt.

Das dazugehörige Unterprogramm kann wie folgt aussehen:

```
C022   18        ADD  CLC         ;Addition vorbereiten
C023   A5 1A          LDA $1A     ;Low-Bytes ...
C025   65 1C          ADC $1C     ;addieren und in ...
C027   85 1E          STA $1E     ;Ergebnisspeicher
C029   A5 1B          LDA $1B     ;High-Bytes ...
C02B   65 1D          ADC $1D     ;addieren und in ...
C02D   85 1F          STA $1F     ;Ergebnisspeicher
C02F   60             RTS
```

Es folgt wieder ein BASIC-Ladeprogramm, mit dem Sie das Programm zur Parameterübergabe durch vereinbarte Speicher eingeben können:

```
1 REM ..... PARAMETERUEBERGABE (VEREINBARTE SPEICHER)
10 DATA 173,  3,200,133, 26,173,  4,200,133, 27
20 DATA 173,  5,200,133, 28,173,  6,200,133, 29
30 DATA  32, 34,192,165, 30,141,  1,200,165, 31
40 DATA 141,  2,200, 96, 24,165, 26,101, 28,133
50 DATA  30,165, 27,101, 29,133, 31, 96
60 FOR I = 49152 TO 49199
70 READ K: POKE I,K
80 SUM=SUM+K
90 NEXT I
100 IF SUM <> 4671 THEN PRINT "PRUEFSUMME FALSCH!":END
110 PRINT "PRUEFSUMME RICHTIG!"
```

Das Programm wird von BASIC aus wieder mit SYS 49152 aufgerufen und kann ähnlich wie die vorigen Programme getestet werden.

Das soeben besprochene Unterprogramm benutzt den Akkumulator als Zwischenspeicher für die zu addierenden Zahlen. Wenn im aufrufenden Programm der Akkumulator ebenfalls benötigt wird, muß der Inhalt des Akkumulators folglich "gerettet" werden: eine einfache Möglichkeit besteht darin, den Inhalt des Akkumulators in den Stack zu bringen.

Damit sieht unser Unterprogramm nun wie folgt aus:

```
CO22   48         ADD  PHA        ;Akku-Inhalt retten
CO23   18              CLC        ;Addition vorbereiten
CO24   A5 1A           LDA $1A    ;Low-Bytes ...
CO26   65 1C           ADC $1C    ;addieren und in ...
CO28   85 1E           STA $1E    ;Ergebnisspeicher
CO2A   A5 1B           LDA $1B    ;High-Bytes ...
CO2C   65 1D           ADC $1D    ;addieren und in ...
CO2E   85 1F           STA $1F    ;Ergebnisspeicher
CO30   68              PLA        ;Akku-Inhalt in ...
                                  ;den Stack zurück
CO31   60              RTS
```

Entsprechend wie in diesem Beispiel kann man natürlich auch noch die Inhalte der X- und Y-Register retten. Dabei muß auf Grund der Stack-Organisation auf die Reihenfolge der Speicherung geachtet werden. Unter Verwendung der X- und Y-Register im Unterprogramm können der Anfang und das Ende des Unterprogramms jetzt so aussehen:

```
48        PHA
8A        TXA
48        PHA
98        TYA
48        PHA
...
...
...

68        PLA
A8        TAY
68        PLA
AA        TAX
68        PLA
60        RTS
```

Sicherlich hatten Sie keine Schwierigkeiten, zu erkennen, daß der Befehl TAX (Transfer Accumulator into X) dazu dient, den Inhalt des Akkumulators in das Index-Register X zu übertragen.

TAX

Je ausgiebiger man den Stack als Zwischenspeicher benutzt, desto weniger Unterprogramme kann man aufrufen. Es gibt natürlich auch andere Möglichkeiten zur Zwischenspeicherung von Registerinhalten, die den Stack nicht benutzen: man muß lediglich bestimmte Speicherbereiche vereinbaren, die die Inhalte von Registern zwischenspeichern sollen. Der Vorteil eines solchen frei definierten Speicherbereiches liegt darin, daß man im Unterprogramm auch noch ohne Schwierigkeiten auf alte Registerinhalte zugreifen kann. Nachteilig ist, daß mehr Speicherplatz und Zeit benötigt wird. Die nächste Befehlsfolge zeigt eine Realisierung der soeben beschriebenen Möglichkeit:

```
8D OC O3        STA $030C
8E OD O3        STX $030D
8C OE O3        STY $030E
...
...
...

AD OC O3        LDA $030C
AE OD O3        LDX $030D
AC OE O3        LDY $030E
6O              RTS
```

Hier wird vereinbart, daß für das Zwischenspeichern des Akkumulators, des X-Registers und des Y-Registers die Speicherstellen mit den Adressen $030C, $030D und $030E verwendet werden. Selbstverständlich ist bei dieser Art der Adressierung die Reihenfolge, in der die Daten zurückgeholt werden, beliebig.

5.1.4 Direkte Parameterübergabe

Die direkte Parameterübergabe wird hier nur der Vollständigkeit halber aufgeführt. Da sie im Kapitel über die 16-Bit-Routinen besprochen wird, verzichten wir hier auf ein zusätzliches Beispielprogramm.

6 | 16-Bit-Simulationen

6. 16-Bit-Simulationen

Die meisten Berechnungen bei der Grafik-Programmierung sind 16-Bit-Berechnungen. Deshalb sollen nun verschiedene Routinen entwickelt werden, mit denen wir 16-Bit-Daten manipulieren können. Diese Routinen, also Unterprogramme, simulieren damit eine Fähigkeit, die der Prozessor, der ja gleichzeitig jeweils nur 8 Bit verarbeiten kann, gar nicht besitzt.

Natürlich ist es nicht notwendig, alle 8-Bit-Befehle des 6510 zu simulieren, sondern nur diejenigen, die man sehr oft benötigt, wie z. B. Addieren, Subtrahieren, Inkrementieren, Dekrementieren, indirekt Laden und Speichern usw.

Um die Programmausführung zu beschleunigen, müssen wir unsere "Register" in der Zero Page unterbringen. Dazu belegen wir folgende Adressen:

$A3 und $A4 dienen als "Generalregister" (GREG), gewissermaßen als "16-Bit-Akkumulator". $FB - $FE stellen einen "Kommunikations-Bereich" dar. Diese Speicheradressen wurden deshalb ausgewählt, weil sie vom Prozessor ohnehin nicht benutzt werden. FNC ($FB) wird zum Speichern einer Funktionsnummer benutzt. PAR ($FD - $FE) ist ein 2-Byte-Register für die Parameterspeicherung.

Für neun weitere Register (RO - R7 und RSAV) wird der Bereich ab $4B benutzt, der laut Commodore-Reference-Guide für temporäre Pointer und Daten verwendet wird. (Es kann allerdings Probleme geben, wenn Sie gleichzeitig BASIC-Programme benutzen. In dem Fall sollten Sie hier auf einen anderen Speicherbereich ausweichen und z.B. den Kassettenpuffer verwenden, der bei $33C - dezimal 828 - beginnt.) Voraussetzung für die Wahl dieses Adressbereiches war, daß es sich um einen zusammenhängenden Speicherbereich handeln mußte.

REGADR ($FC) wird zum Speichern von Registeradressen - d.h. der Indizes von 0 bis 7 - benutzt. Hierbei dient die höherwertige Byte-Hälfte zur Speicherung des Quell-Registers und die niederwertige zur Speicherung des Ziel-Registers (bei Operationen mit zwei Registern).

BITMAP ($D011, 53265 dezimal) ist die Adresse des Video-Chip-Kontrollregisters und SCRMEM ($D018, 53272 dezimal) die Adresse des Video-Chip-Speicher-Kontroll-registers. Bei PAG1 ($100) beginnt der Bereich des Stack-Speichers, der bis $1FF reicht.

GREG	$A3	$A4
R0	$4B	$4C
R1	R0+2	R0+3
R2	R1+2	R1+3
R3	R2+2	R2+3
R4	R3+2	R3+3
R5	R4+2	R4+3
R6	R5+2	R5+3
R7	R6+2	R6+3

Abb.: "16-Bit"-Register

Diese Speicherplätze benutzen wir jetzt ähnlich, wie der Prozessor seine internen Register benutzt, mit dem Unterschied, daß wir jeweils zwei Bytes zusammen als ein 16-Bit-Register betrachten. Damit wir die einzelnen "16-Bit-Befehle" bequem auswählen können, werden sie in eine Tabelle eingetragen, so daß zu ihrem Aufruf nur ihre Position in dieser Tabelle angegeben werden muß.

Die Tabelle enthält 2-Byte-Funktionen (Nr. 1-31) und 4-Byte-Funktionen (ab Nr. 32). Bei den 2-Byte-Funktionen sind die Eintragungen nach der Anzahl der Register, die manipuliert werden, sortiert: Operationen mit einem Register, mit zwei Registern und mit mehreren Registern.

Dazu kommen natürlich noch die Parameter, die übergeben werden müssen. Die Parameter-Übergabe erfolgt hier direkt über das aufrufende Benutzer-Programm: die Daten, also die Funktionsnummer, die Registeradresse(n) und der Parameter, der der betreffenden Funktion übergeben werden soll, stehen im Benutzerprogramm nach dem Aufruf des noch zu besprechenden 16-Bit-Interpreters.

ADRESS-TABELLE DER 16-BIT-ROUTINEN

```
83:    C042 E6 C0   FNCTAB   .WORDCLAREG-1   ;1 CLEAR ALL REGISTERS
84:    C044 F0 C0            .WORDCLREG-1    ;2 CLEAR REGISTER
85:    C046 FA C0            .WORDINCREG-1   ;3 INCREMENT REGISTER
86:    C048 04 C1            .WORDDECREG-1   ;4 DECREMENT REGISTER
87:    C04A 14 C1            .WORDSRREG-1    ;5 SHIFT RIGHT REGISTER
88:    C04C 22 C1            .WORDSLREG-1    ;6 SHIFT LEFT REGISTER
89:    C04E 30 C1            .WORDCSNREG-1   ;7 CHANGE SIGN OF REGISTER
90:    C050 40 C0            .WORDDOIT-1     ;8
91:    C052 40 C0            .WORDDOIT-1     ;9
92:    C054 40 C0            .WORDDOIT-1     ;10
93:    C056 41 C1            .WORDTRAREG-1   ;11 TRANSFER REGISTER
94:    C058 50 C1            .WORDCMPREG-1   ;12 COMPARE REGISTER
95:    C05A 62 C1            .WORDADD-1      ;13 ADDITION
96:    C05C 76 C1            .WORDSUB-1      ;14 SUBTRACTION
97:    C05E 40 C0            .WORDDOIT-1     ;15
98:    C060 40 C0            .WORDDOIT-1     ;16
99:    C062 40 C0            .WORDDOIT-1     ;17
100:   C064 40 C0            .WORDDOIT-1     ;18
101:   C066 40 C0            .WORDDOIT-1     ;19
102:   C068 40 C0            .WORDDOIT-1     ;20
103:   C06A 40 C0            .WORDDOIT-1     ;21
104:   C06C 40 C0            .WORDDOIT-1     ;22
105:   C06E 40 C0            .WORDDOIT-1     ;23
106:   C070 40 C0            .WORDDOIT-1     ;24
107:   C072 40 C0            .WORDDOIT-1     ;25
108:   C074 40 C0            .WORDDOIT-1     ;26
109:   C076 40 C0            .WORDDOIT-1     ;27
110:   C078 40 C0            .WORDDOIT-1     ;28
111:   C07A 40 C0            .WORDDOIT-1     ;29
112:   C07C 40 C0            .WORDDOIT-1     ;30
113:   C07E 40 C0            .WORDDOIT-1     ;31
114:   C080 8D C1            .WORDSETREG-1   ;32 SET REGISTER
115:   C082 99 C1            .WORDLRIND-1    ;33 LOAD REGISTER INDIRECT
116:   C084 A7 C1            .WORDSTOREG-1   ;34 STORE REGISTER INDIRECT
117:   C086 40 C0            .WORDDOIT-1     ;35
```

6.1 Häufig verwendete 16-Bit-Routinen

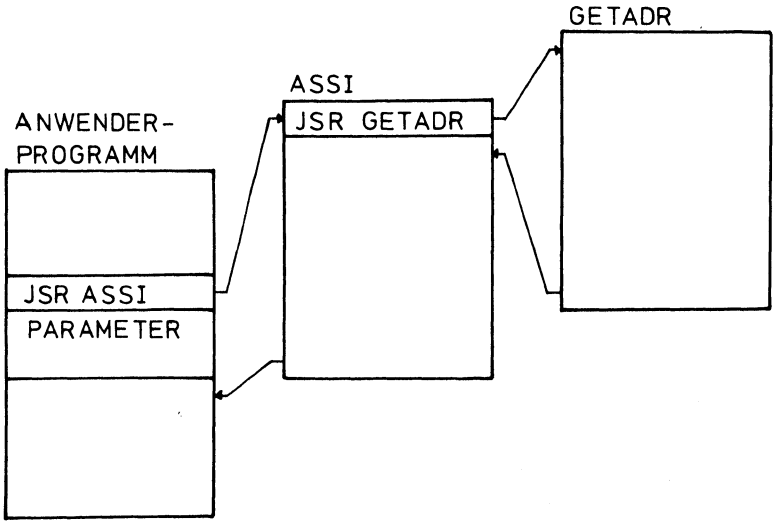
```
                    ;ALLGEMEIN VERWENDETE ROUTINEN FUER DEN 16-BIT-INTERPRETER


                    ;RUECKSPRUNGADRESSE HOLEN (GET ADDRESS)
                    ;RETURN MIT RUECKSPRUNGADRESSE IN GREG
125:  C088 BA       GETADR  TSX             ;AKT. STACKPOINTER
126:  C089 E8               INX
127:  C08A E8               INX
128:  C08B E8               INX             ;POINTER FUER RUECKSPRUNGADRESSE -1
129:  C08C BD 00 01         LDA  PAG1,X     ;RUECKSPRUNGADRESSE -1 ...
130:  C08F 85 A3            STA  GREG
131:  C091 E8               INX
132:  C092 BD 00 01         LDA  PAG1,X
133:  C095 85 A4            STA  GREG+1     ;INS GENERALREGISTER

                    ;INCREMENT GENERALREGISTER
136:  C097 E6 A3    INCG    INC  GREG       ;ZUERST DAS LOW-BYTE
137:  C099 D0 02            BNE  INCGX      ;LOW-BYTE <> 0, FERTIG!
138:  C09B E6 A4            INC  GREG+1     ;HIGH-BYTE AUCH INKREMENTIEREN
139:  C09D 60       INCGX   RTS
```

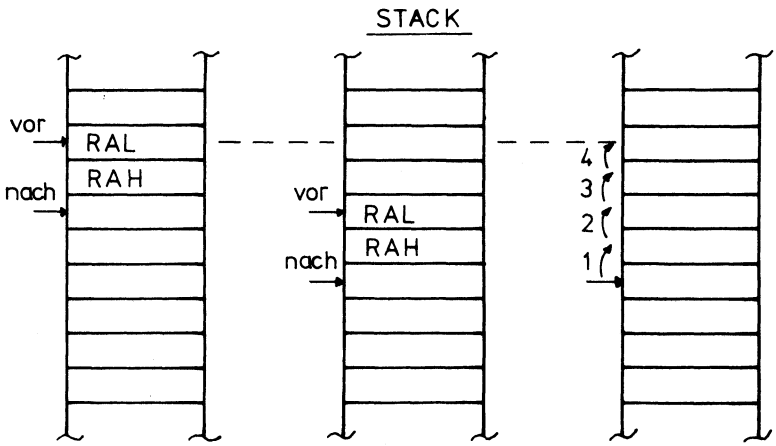
Die Routine GETADR holt eine Rücksprungadresse, d.h. die Adresse, an der ein Programm nach einem Unterprogrammaufruf fortgesetzt werden soll. Die Rücksprungadresse steht anschließend im Register GREG zur Verfügung.

Um an diese Adresse zu gelangen, müssen wir den Stack-Pointer holen (125).

Die folgenden Abbildungen zeigen die Beziehungen zwischen dem aufrufenden Programm und den aufgerufenen Unterprogrammen sowie die Position des Stack-Pointers vor und nach dem Unterprogrammaufruf.

Der linke Teil stellt das Anwender-Programm dar, von dem aus der Aufruf der Routine ASSI erfolgt (s. Kapitel 6.2). Der mittlere Teil der Abbildung steht für das Unterprogramm ASSI, das seinerseits das Unterprogramm GETADR aufruft, d.h. unser Stack-Pointer rutscht noch um eine Position weiter.





RAL = Rückspr.-Adr. Low
RAH = „ „ High

Beim Aufruf von GETADR ist der Zeiger also um drei Bytes versetzt, so daß zum Zurückholen der Adresse ein dreimaliges Inkrementieren erforderlich ist (126-128), wodurch man das Low-Byte der Rücksprungadresse erhält (129), das nach GREG gespeichert wird (130). Durch nochmaliges Inkrementieren erhält man das High-Byte, das in GREG+1 abgelegt wird (131-133).

Um an die eigentliche Adresse zu kommen, müssen wir den Inhalt des General-Registers noch inkrementieren, d.h. um eins erhöhen (136-139). Der Befehl zum Inkrementieren heißt INC (INCrement memory by one). Danach steht in GREG (und GREG+1) die Anfangsadresse unseres Parameters, d.h. der Funktionsnummer. Damit ist das Korrigieren der Rücksprungadresse beendet.

INC

Zunächst wird das Low-Byte von GREG inkrementiert (136). Ist dieser Wert anschließend nicht gleich Null, so wird das Unterprogramm verlassen (137). Ist der Inhalt von GREG gleich Null, so muß auch das High-Byte (GREG + 1) inkrementiert werden (138). Dies soll noch näher erläutert werden:

Angenommen, ein Register enthält die folgenden Werte: High-Byte = 1 (tatsächlich also 256) und Low-Byte = 255, d.h. insgesamt 511.

```
256     255
$01     $FF
```

Wird dieser Wert inkrementiert, muß also 512 herauskommen. Wird $FF inkrementiert, so wird das Low-Byte gleich 0, d.h. es muß ein Übertrag stattfinden. Deshalb wird in der Routine INCG abgefragt, ob der Wert des Low-Bytes (GREG) gleich 0 ist, da in dem Fall das High-Byte um eins erhöht werden muß.

Ein anderes Beispiel: ist der Wert des Low-Bytes gleich 10 (allgemein: ungleich 0), so findet kein Übertrag statt, und das High-Byte wird nicht inkrementiert.

```
256     10
$01     $0A
```

Ganz entsprechend erfolgt das dekrementieren des "Akku"-Registers GREG in der Routine DECG (142-146):

```
                    ;DECREMENT GENERALREGISTER
142:   C09E A5 A3   DECG    LDA  GREG    ;ZUERST DAS LOW-BYTE
143:   C0A0 D0 02          BNE  DECG1   ;LOW-BYTE <> 0, NUR LOW-BYTE DEKREMENTIEREN
144:   C0A2 C6 A4          DEC  GREG+1  ;HIGH-BYTE AUCH DEKREMENTIEREN
145:   C0A4 C6 A3   DECG1   DEC  GREG    ;LOW-BYTE DEKREMENTIEREN
146:   C0A6 60            RTS
```

Der Inhalt unseres "16-Bit-Akku"-Registers (GREG für das Low-Byte und GREG+1 für das High-Byte) wird dekrementiert, also um eins vermindert.

Der Akku wird mit dem Low-Byte (GREG) geladen (142) und es wird überprüft, ob dieser Wert gleich Null ist (143). Ist GREG ungleich Null, so wird es dekrementiert, d.h. um eins vermindert, (145) und der Wert des High-Bytes (GREG+1) braucht nicht berücksichtigt zu werden. Der Befehl zum dekrementieren heißt DEC (DECrement memory by one). Ist GREG jedoch gleich Null, so muß auch das High-Byte um eins vermindert werden (144).

DEC

Das Prozessorstatus-Register wird hier nicht aktualisiert.

Angenommen, das High-Byte ist gleich 1 und das Low-Byte gleich 0, der Gesamtwert also 256:

1	0
$01	$00

Der dekrementierte Wert ist 255. Man erhält ihn, indem man das High-Byte und das Low-Byte dekrementiert, da man beim dekrementieren von $00 den Wert $FF (also 255) erhält:

0	255
$00	$FF

Beispiel: DECG

Vorher: GREG: $02FF (GREG: $FF GREG+1: $02)
Nachher: GREG: $02FE (GREG: $FE GREG+1: $02)

```
                    ;RUECKSPRUNGADRESSE IN DEN STACK ZURUECKSCHREIBEN
                    ;RUECKSPRUNGADRESSE -1 IN GREG
150:  C0A7 BA    PUTADR  TSX           ;AKT. STACKPOINTER
151:  C0A8 E8            INX
152:  C0A9 E8            INX
153:  C0AA E8            INX           ;OFFSET DER RUECKSPRUNGADRESSE IM STACK
154:  C0AB A5 A3         LDA  GREG     ;LOW-BYTE DER RUECKSPRUNGADRESSE ...
155:  C0AD 9D 00 01      STA  PAG1,X   ;IN DEN STACK
156:  C0B0 E8            INX
157:  C0B1 A5 A4         LDA  GREG+1   ;HIGH-BYTE
158:  C0B3 9D 00 01      STA  PAG1,X
159:  C0B6 60            RTS
```

Diese Routine schreibt eine Rücksprungadresse in den Stack zurück.

Zunächst wird hier der aktuelle Stack-Pointer nach X transferiert (150). Dann wird dreimal inkrementiert (151-153, vgl. GETADR 126-128). Der Inhalt von GREG wird in den Akku geladen (154) und - durch Zero-Page X-indizierte Adressierung - im Stack gespeichert (155). Ganz entsprechend wird mit dem High-Byte von GREG verfahren (156-158).

Mit PUTADR werden also die aktuellen High- und Low-Bytes von GREG "gerettet", indem sie auf dem Stack zwischengespeichert werden. Auf diese Weise wird die Rücksprungadresse korrigiert.

```
                    ;LADE GENERALREGISTER INDIREKT
                    ;AUFRUF MIT OPERANDENADRESSE IN GREG
                    ;RETURN MIT OPERAND IN GREG
164:  C0B7 A0 00  LGIND   LDY  #$00     ;INDEX=0
165:  C0B9 B1 A3          LDA  (GREG),Y ;LOW-BYTE DES OPERANDEN
166:  C0BB 48             PHA
167:  C0BC C8             INY           ;INDEX=1
168:  C0BD B1 A3          LDA  (GREG),Y ;HIGH-BYTE DES OPERANDEN
169:  C0BF 85 A4          STA  GREG+1
170:  C0C1 68             PLA
171:  C0C2 85 A3          STA  GREG
172:  C0C4 60             RTS
```

Der "Akku" GREG wird mit dem Inhalt des Speichers geladen, dessen Adresse in GREG steht.

Zum indirekten Laden muß die indirekt-indizierte Adressierung mit Y verwendet werden, da es im Befehlssatz des 6510 keinen Befehl zum indirekten Laden gibt. Da Y als Index-Register verwendet werden soll, muß es erst einmal gelöscht werden (164). Dann wird das Low-Byte geholt (165) und in den Stack geschoben (166). Um den Inhalt des High-Bytes zu holen, muß natürlich die Adresse erhöht werden. Dies kann man mit der Inkrementierung des Y-Index-Registers erreichen. Nachdem Y inkrementiert wurde (167), wird das High-Byte geladen (168).

Der bisherige Inhalt von GREG (also die Adresse) wird nun nicht mehr benötigt und kann deshalb überschrieben werden. Der zuletzt geholte Wert, d.h. das High-Byte, kann also sofort in das High-Byte-Register von GREG gespeichert werden (169). Das zuvor in den Stack geschobene Low-Byte wird aus dem Stack geholt (170) und in das Low-Byte-Register von GREG gespeichert (171).

Beispiel: LGIND

Vorher: GREG: $1000 ($1000: $01 $1001: $02)
Nachher: GREG: $0201 (GREG: $01 GREG+1: $02)

```
                ;INDEX DES QUELLENREGISTERS (SOURCE REGISTER) HOLEN
                ;RETURN MIT REGISTERNUMMER*2 IM X-INDEXREGISTER
                ;GET INDEX FOR SRC.-REG.
177:  C0C5 A5 FC  GETSR   LDA  REGADR
178:  C0C7 29 70          AND  #$70     ;REGISTERNUMMER DES ZIELREGISTERS AUSBLENDEN
179:  C0C9 4A             LSR
180:  C0CA 4A             LSR
181:  C0CB 4A             LSR           ;3X NACH RECHTS SCHIEBEN = REGISTERNUMMER*2
182:  C0CC AA             TAX
183:  C0CD 60             RTS
```

Mit diesem Unterprogramm holt man den Index eines Quellen-Registers.

Der Akku wird mit der Registeradresse REGADR geladen (177). Durch die UND-Verknüpfung mit $70 wird das Low-Nibble, d.h. die untere Byte-Hälfte, ausgeblendet (178).

Mit dem Befehl LSR (Logical Shift Right) wird jetzt das High-Nibble (also die vier oberen Bits) zum Low-Nibble verschoben. LSR schiebt den Inhalt des Akkus oder eines Speichers um eine Bitstelle nach rechts. Dabei erhält Bit 7 den Wert 0, und Bit 0 wird in das Carry-Bit C geschoben.

LSR

Eigentlich müßte dieser Befehl viermal ausgeführt werden, damit die Bits um vier Positionen verschoben werden. Tatsächlich erfolgt hier jedoch nur ein dreimaliges Verschieben (179-181). Das ist so zu erklären:

Die Register R1 - R7 werden jeweils relativ zu R0 angesprochen, d.h. wenn Register R1 gemeint ist, wird es durch R0+2 und R0+3 angesprochen. Dies ist ja möglich, da die Adressen dieser Register direkt aufeinander folgen. Um z.B. R3 anzusprechen, muß man also die sechs Bytes überspringen, die von den drei ersten Registern (R0 - R2) belegt werden, d.h. R0+6 und (R0+1)+6 für die zweite Registerhälfte.

```
R0

R1   =    R0+2          Low-Byte
        (R0+1)+2        High-Byte

R2   =    R0+4          Low-Byte
        (R0+1)+4        High-Byte
...
...
...

R7   =    R0+14         Low-Byte
        (R0+1)+14       High-Byte
```

Allgemein muß immer der doppelte Wert der Registernummer zu R0 hinzuaddiert werden. Genau dies erreicht man durch das dreimalige Verschieben nach rechts: das Ergebnis ist zweimal so groß wie bei der viermaligen Anwendung von LSR.

Der so veränderte Akku-Inhalt wird ins X-Register kopiert (184). GETSR bringt also die mit zwei multiplizierte Registernummer ins X-Register.

```
                    ;INDEX DES ZIELREGISTERS (DESTINATION REGISTER) HOLEN
                    ;RETURN MIT REGISTERNUMMER*2 IM X-INDEXREGISTER
                    ;GET INDEX FOR DST.-REG.
188:  C0CE A5 FC  GETDI   LDA  REGADR
189:  C0D0 29 07          AND  #$07     ;REGISTERNUMMER DES QUELLENREGISTERS AUSBLENDEN
190:  C0D2 0A             ASL           ;1X NACH LINKS SCHIEBEN = REGISTERNUMMER*2
191:  C0D3 AA             TAX
192:  C0D4 60             RTS
```

Durch das Unterprogramm GETDI wird der Index für das Zielregister geholt. Dazu wird der Inhalt von REGADR in den Akku gebracht (188). Durch die UND-Verknüpfung mit 7 wird das High-Nibble dieses Bytes ausgeblendet (189).

Das Low-Nibble wird mit ASL (Arithmetic Shift Left) um ein Bit nach links verschoben (190). Dabei erhält Bit 0 den Wert 0, und Bit 7 (MSB) wird in das Carry-Flag C geschoben. Unter Berücksichtigung des Übertrags wird also der Inhalt des angegebenen Speichers verdoppelt.

ASL

Der so verdoppelte Wert wird ins X-Register gebracht, um als Index zu dienen (191).

Wie Sie sehen, werden überwiegend 1- oder 2-Byte-Befehle benutzt, damit die Programme sowohl schneller als auch kürzer werden.

```
                    ;LADE GENERALREGISTER MIT SOURCEREGISTER
                    ;AUFRUF MIT INDEX DES 16-BIT-REGISTERS IN REGADR
                    ;  ODER MIT INDEX DES 16-BIT-REGISTERS IM X-INDEXREGISTER,
                    ;       WOBEI LOADR DIE EINSPRUNGSTELLE IST
                    ;LOAD GENERALREG. WITH SRC.-REG
199:  C0D5 20 C5 C0 LOADS   JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
200:  C0D8 B5 4B    LOADR   LDA  R0,X     ;LOW-BYTE
201:  C0DA 85 A3            STA  GREG
202:  C0DC B5 4C            LDA  R0+1,X   ;HIGH-BYTE
203:  C0DE 85 A4            STA  GREG+1
204:  C0E0 60                RTS
```

Der Inhalt eines bestimmten Source- (Quellen-) Registers wird nach GREG geladen.

Der von GETSR ins X-Register gebrachte Wert (199) dient jetzt als Index, um den Akku mit dem Inhalt des gewünschten Registers (R0 - R7) zu laden (200). Anschließend wird dieser Wert - zunächst das Low-Byte - nach GREG gespeichert (201). Entsprechend wird das High-Byte des Registers nach GREG+1 gebracht (202-203).

```
                    ;LADE GENERALREGISTER MIT ZIELREGISTER
                    ;AUFRUF MIT INDEX DES 16-BIT-REGISTERS IN REGADR
208:  C0E1 20 CE C0 LOADD   JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
209:  C0E4 4C D8 C0         JMP  LOADR
```

Die Routine LOADD lädt das General-Register mit dem Ziel-Register.

Mit GETDI wird der Index des Ziel-Registers geholt (208). Der weitere Ablauf entspricht genau dem Laden von GREG mit dem Quell-Register (LOADS), so daß der entsprechende Programmteil wieder benutzt werden kann. (209: 200-204). Der Befehl JMP (JuMP to new location) bewirkt die Verzweigung zu der angegebenen Adresse.

JMP

```
                    ;ALLE REGISTER LOESCHEN (CLEAR ALL REGISTERS)
212:  C0E7 A9 00    CLAREG  LDA  #$00
213:  C0E9 A2 10            LDX  #$10    ;16 BYTES
214:  C0EB 95 4B    CLR     STA  R0,X
215:  C0ED CA               DEX
216:  C0EE 10 FB            BPL  CLR
217:  C0F0 60               RTS
```

Alle Register (R0 bis R7) werden gelöscht.

Zum Löschen wird der Akku mit dem Wert 0 geladen (212). Das Index-Register X wird mit $10 (also dezimal 16) geladen, da wir 2*8=16 Bytes löschen müssen (213).

In einer Programm-Schleife wird dann der Akku-Inhalt in die einzelnen Register gebracht (214-216), wobei der Wert von X jedesmal dekrementiert wird (215). Dazu wird der Dekrementierbefehl DEX (DEcrement index X by one) verwendet. Die Schleife wird solange durchlaufen, wie X positiv ist, wobei die Null auch als positiver Wert gilt (216). Diese Bedingung wird mit dem Befehl BPL (Branch on result PLus) getestet.

DEX

BPL

```
                    ;REGISTER LOESCHEN (CLEAR REGISTER)
220:  C0F1 20 C5 C0 CLREG   JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
221:  C0F4 A9 00            LDA  #$00
222:  C0F6 95 4B            STA  R0,X    ;LOW-BYTE
223:  C0F8 95 4C            STA  R0+1,X  ;HIGH-BYTE
224:  C0FA 60               RTS
```

Mit dieser Routine kann der Inhalt eines bestimmten Registers (RO bis R7) gelöscht werden.

Mit GETSR wird der Index des Quell-Registers geholt (220). Zum Löschen wird der Wert Null in den Akku gebracht (221) und anschließend in das Low- und High-Byte des zu löschenden Registers (222-223).

```
                    ;INCREMENT REGISTER
227:  C0FB 20 C5 C0 INCREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
228:  C0FE F6 4B            INC  R0,X
229:  C100 D0 02            BNE  INRX
230:  C102 F6 4C            INC  R0+1,X
231:  C104 60       INRX    RTS
```

Der Inhalt eines bestimmten Registers (RO bis R7) wird inkrementiert, also um eins erhöht.

Zunächst wird der Index des betreffenden Registers mit GETSR geholt (227). Mit diesem Index indiziert wird dann das Low-Byte des Registers inkrementiert (228). Ähnlich wie bei der Routine INCG (136-139) wird dann abgefragt, ob das Low-Byte gleich Null ist (229). Ist dies der Fall, so wird auch das High-Byte inkrementiert (230).

```
                    ;DECREMENT REGISTER
234:  C105 20 C5 C0 DECREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
235:  C108 B5 4B            LDA  R0,X
236:  C10A D0 02            BNE  DECR1
237:  C10C D6 4C            DEC  R0+1,X
238:  C10E D6 4B    DECR1   DEC  R0,X
239:  C110 B5 4B            LDA  R0,X    ;KONDITIONSBIT (ZERO FLAG) AKTUALISIEREN
240:  C112 15 4C            ORA  R0+1,X
241:  C114 60               RTS
```

Der Inhalt eines bestimmten Registers (RO bis R7) wird dekrementiert, also um eins vermindert.

Sofern das Low-Byte des betreffenden Registers ungleich Null ist (235-236), wird nur das Low-Byte dekrementiert (238). Ist der Wert jedoch gleich Null, müssen beide Bytes des Registers dekrementiert werden (237-238).

Nun müssen noch das N- und das Z-Flag geändert werden: dazu wird das Low-Byte des Registers in den Akku geladen, dessen Inhalt dann durch ORA ("OR" memory with Accumulator) mit dem High-Byte des Registers verknüpft wird (239-240). Es handelt sich hierbei um das einschließende logische ODER. Das Ergebnis dieser Verknüpfung ist dann logisch "wahr" (Ergebnis = 1), wenn nur einer oder auch beide Operanden den Wert 1 haben. Die folgende Tabelle enthält die möglichen Kombinationen für zwei Operanden:

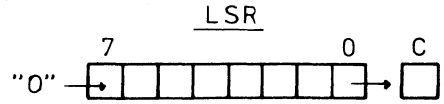
ORA

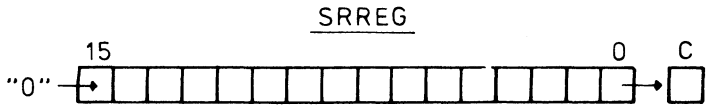
A	B	A ODER B
1	1	1
1	0	1
0	1	1
0	0	0

```
                        ; SHIFT RIGHT REGISTER
244:  C115 20 C5 C0 SRREG   JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
245:  C118 B5 4C            LDA  R0+1,X
246:  C11A 4A               LSR          ;SHIFT RIGHT HIGH-BYTE, MSB IN CARRY
247:  C11B 95 4C            STA  R0+1,X  ;ZURUECKSCHREIBEN
248:  C11D B5 4B            LDA  R0,X    ;LOW-BYTE
249:  C11F 6A               ROR          ;ROTIEREN NACH RECHTS DURCH CARRY
250:  C120 95 4B            STA  R0,X    ;ZURUECKSCHREIBEN
251:  C122 60               RTS
```

Mit dieser Routine wird der Inhalt des angegebenen Registers (RO bis R7) um eine Bitstelle nach rechts verschoben.

Dieses Unterprogramm bildet den 6510-Befehl LSR nach, der den Inhalt des Akkus oder eines Speichers um eine Bitstelle nach rechts schiebt. Unter Berücksichtigung des Übertrags wird also der Inhalt des Registers (ganzzahlig, ohne Rest) halbiert.





Mit GETSR wird der Index des betreffenden Registers geholt (244). Das High-Byte dieses Registers wird in den Akku geladen (245), um eine Bitstelle nach rechts verschoben (246) und wieder zurück in das Register gebracht (247).

Um den LSR-Befehl nachzubilden, muß das augenblicklich im C-Flag befindliche Bit das höchstwertige Bit des Low-Bytes werden, dessen übriger Inhalt natürlich ebenfalls um eine Bitstelle nach rechts verschoben werden muß.

Dazu verwenden wir - nachdem das Low-Byte in den Akku gebracht wurde (248) - den Befehl ROR (ROtate Right one bit), der den Inhalt des Akkus oder einer anderen Speicherstelle um ein Bit durch das Carry-Flag nach rechts rotiert (249). Dabei kommt der Inhalt des C-Flags nach Bit 7 und Bit 0 nach C, so daß der 9-Bit-Inhalt von Register-Low-Byte und C-Flag um eine Bitstelle nach rechts rotiert.

ROR

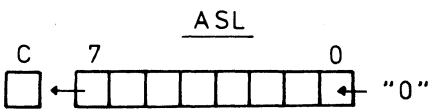
Anschließend wird auch das High-Byte wieder in das entsprechende Register zurückgeschrieben (250). Auf diese Weise erreichen wir für unser 16-Bit-Register insgesamt ein Verschieben nach rechts, wie es dem 8-Bit-Befehl LSR entspricht.

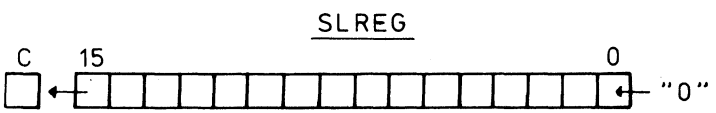
```
                        ; SHIFT LEFT REGISTER
254:   C123 20 C5 C0 SLREG    JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
255:   C126 B5 4B            LDA  R0,X     ;LOW-BYTE
256:   C128 0A               ASL           ;SHIFT LEFT LOW-BYTE, MSB IN CARRY
257:   C129 95 4B            STA  R0,X     ;ZURUECKSCHREIBEN
258:   C12B B5 4C            LDA  R0+1,X   ;HIGH-BYTE
259:   C12D 2A               ROL           ;ROTIEREN NACH LINKS DURCH CARRY
260:   C12E 95 4C            STA  R0+1,X   ;ZURUECKSCHREIBEN
261:   C130 60               RTS
```

Mit dieser Routine wird der Inhalt des angegebenen Registers (RO bis R7) um eine Bitstelle nach links verschoben.

Dieses Unterprogramm bildet den Befehl ASL nach, der den Inhalt des Akkus oder eines anderen Speichers um eine Bitstelle nach links verschiebt. Unter Berücksichtigung des Übertrags wird also der Akku- bzw. Speicherinhalt verdoppelt.

Analog zu dem 8-Bit-Befehl ASL wird mit SREGL eine entsprechende Operation mit 16 Bit ausgeführt, wobei Bit O des Registers den Wert O erhält und Bit 15 (MSB) in das Carry-Flag geschoben wird.





Mit GETSR wird der Index des betreffenden Registers geholt (254). Das Low-Byte dieses Registers wird in den Akku geladen (255), um eine Bitstelle nach links verschoben (256) und wieder zurück in das Register gebracht (257).

Wir müssen nun dafür sorgen, daß das "herausgefallene" Bit 7 des Low-Bytes zum Bit 0 des High-Bytes wird und außerdem zuvor alle Bits des High-Bytes um eine Bitstelle nach links verschoben werden. Dafür steht der Rotations-Befehl ROL (ROtate Left one bit) zur Verfügung, der den Inhalt der angegebenen Speicherstelle oder des Akkus um ein Bit nach links durch das Carry-Flag rotiert. Dabei kommt der Inhalt des C-Flags nach Bit 0 und der Inhalt von Bit 7 ins C-Flag, so daß also die insgesamt 9 Bits (Register-High-Byte und Übertragsbit) um eine Stelle rotieren (258-259). Das Ergebnis wird wieder zurück in das High-Byte des Registers gebracht (260).

ROL

Auf diese Weise erreichen wir für unser 16-Bit-Register insgesamt ein Verschieben nach links, wie es dem 8-Bit-Befehl ASL entspricht.

```
                        ;REGISTER-VORZEICHEN WECHSELN (CHANGE SIGN OF REGISTER)
264:  C131 20 C5 C0 CSNREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
265:  C134 38               SEC          ;SUBTRAKTION VORBEREITEN
266:  C135 A9 00            LDA  #$00    ;NULL ...
267:  C137 F5 4B            SBC  R0,X    ;MINUS REGISTER
268:  C139 95 4B            STA  R0,X    ;ZURUECKSCHREIBEN
269:  C13B A9 00            LDA  #$00    ;DESGLEICHEN FUER HIGH-BYTE
270:  C13D F5 4C            SBC  R0+1,X
271:  C13F 95 4C            STA  R0+1,X
272:  C141 60               RTS
```

Mit dieser Routine wird das Vorzeichen eines bestimmten Registers (R0 bis R7) geändert.

Der Vorzeichenwechsel wird dadurch erreicht, daß der Inhalt des betreffenden Segisters von Null subtrahiert wird. Nachdem mit GETSR der Register-Index geholt worden ist (264), werden die Low- und High-Bytes jeweils von Null subtrahiert (265-271).

Ebenso wie bei der Addition ist auch eine Subtraktion beim 6510 nur mit Übertrag möglich. Der Subtraktionsbefehl SBC (SuBtract with Carry) subtrahiert den Inhalt der angegebenen Adresse (oder Konstanten) sowie den komplementierten Wert des Carry-Bits vom Akkumulator. Um einen Übertrag zu vermeiden, muß also das Carry-Bit vor der Subtraktion mit dem Befehl SEC (SEt Carry) gesetzt werden (265).

SBC

SEC

```
                    ;QUELLENREGISTER -> ZIELREGISTER (TRANSFER REGISTER)
                    ;INHALT DES QUELLENREGISTERS WIRD NICHT GEAENDERT
276:  C142 20 D5 C0 TRAREG  JSR  LOADS   ;GREG MIT INHALT DES QUELLENREGISTERS LADEN
277:  C145 20 CE C0 SAVG    JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
278:  C148 A5 A3    SAVREG  LDA  GREG    ;INHALT DES GENERALREGISTERS INS GEWUENSCHTE
279:  C14A 95 4B            STA  R0,X    ;REGISTER SPEICHERN
280:  C14C A5 A4            LDA  GREG+1
281:  C14E 95 4C            STA  R0+1,X
282:  C150 60               RTS
```

Diese Routine kopiert den Inhalt eines Registers (Quelle) in ein anderes (Ziel).

Zunächst wird der Inhalt des Quell-Registers geholt (276), dann die Ziel-Adresse (277). Danach wird der Inhalt von GREG ins Ziel-Register kopiert (278-279). Entsprechend wird mit dem High-Byte von GREG verfahren (280-281).

Beispiel: TRAREG

	Vorher:	Nachher:
Reg.-Adr.	$01	$01
R0	$1000	$1000
R1	$xxxx	$1000

```
                    ;REGISTER VERGLEICHEN (COMPARE REGISTER)
                    ;DIE KONDITIONSBITS WERDEN WIE BEIM 8-BIT-BEFEHL AKTUALISIERT
286:  C151 20 C5 C0 CMPREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM AKKU ...
287:  C154 A8               TAY          ;UND JETZT IM Y-INDEXREGISTER
288:  C155 20 CE C0         JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
289:  C158 B9 4B 00         LDA  R0,Y    ;JETZT VERGLEICHEN
290:  C15B D5 4B            CMP  R0,X
291:  C15D B9 4C 00         LDA  R0+1,Y
292:  C160 F5 4C            SBC  R0+1,X
293:  C162 60               RTS
```

Die Inhalte zweier Register (RO bis R7) werden miteinander verglichen.

Da nach dem Aufruf von GETSR der Register-Index sowohl in X als auch im Akku steht (286), kann er vom Akku gleich nach Y gebracht werden (287).

Mit GETDI wird dann der Index des zweiten Registers geholt (288), dessen Low-Byte anschließend im X-Register und im Akku steht.

Der Akku wird nun mit dem Inhalt von RO,Y (also dem Low-Byte des ersten Registers) geladen (289) und mit dem Low-Byte des zweiten Registers verglichen (290). Dies geschieht mit dem Vergleichsbefehl CMP (CoMPare memory and accumulator). Dem Ergebnis des Vergleichs entsprechend werden die drei Flags N, Z und C verändert: Z = 1, wenn beide Werte gleich sind; N = 1, wenn der Akku-Inhalt kleiner ist als der Vergleichswert und C = 1, wenn der Akku-Inhalt größer oder gleich dem Vergleichswert ist.

CMP

Nachdem dann der Akku mit dem High-Byte des ersten Registers geladen wurde (291), wird das High-Byte des zweiten Registers vom Akku subtrahiert (292).

```
                    ;ADDITION VON QUELLEN- UND ZIELREGISTER
                    ;ERGEBNIS IM QUELLENREGISTER, ZIELREG. BLEIBT UNVERAENDERT
297:  C163 20 E1 C0 ADD     JSR  LOADD   ;LADE GENERALREGISTER MIT ZIELREGISTER
298:  C166 20 C5 C0         JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
299:  C169 18               CLC          ;ADDITION VORBEREITEN
300:  C16A B5 4B            LDA  R0,X    ;LOW-BYTE DES QUELLENREGISTERS ZUM ...
301:  C16C 65 A3            ADC  GREG    ;LOW-BYTE DES GENERALREGISTERS ADDIEREN ...
302:  C16E 95 4B            STA  R0,X    ;UND INS QUELLENREGISTER SPEICHERN
303:  C170 B5 4C            LDA  R0+1,X  ;DAS GLEICHE AUCH FUER DAS HIGH-BYTE
304:  C172 65 A4            ADC  GREG+1
305:  C174 95 4C            STA  R0+1,X
306:  C176 60               RTS
```

Die Inhalte von Quell- und Ziel-Register werden addiert. Das Ergebnis wird im Quell-Register gespeichert.

Zunächst wird durch den Aufruf von LOADD (297) das General-Register mit dem Ziel-Register (Destination) geladen.

Dann wird durch das Unterprogramm GETSR der Index für das Quell-Register (Source) geholt (298).

Vor der Addition wird noch das Carry-Bit gelöscht (299). Der Akku wird dann mit dem Wert RO,X geladen, also mit dem Low-Byte des Quell-Registers (da ja in X der Index des Quell-Registers steht) (300). Dazu wird das Low-Byte von GREG addiert, in dem das Ziel-Register steht (301). Das Ergebnis wird im Quell-Register gespeichert (302). Entsprechend erfolgt die Addition der beiden High-Bytes (303-305).

Beispiel: ADD

	Vorher:	Nachher:
Reg.-Adr.	$12	$12
R1	$10F0	$12BA
R2	$01CA	$01CA

```
                        ;ZIELREGISTER VON QUELLENREGISTER SUBTRAHIEREN
                        ;ERGEBNIS IM QUELLENREGISTER, ZIELREG. BLEIBT UNVERAENDERT
310:  C177 20 CE C0 SUB      JSR  GETDI    ;INDEX DES ZIELREGISTERS ...
311:  C17A A8                TAY           ;INS Y-INDEXREGISTER
312:  C17B 20 C5 C0          JSR  GETSR    ;INDEX DES QUELLENREGISTERS INS X-INDEXREGISTER
313:  C17E 38                SEC           ;SUBTRAKTION VORBEREITEN
314:  C17F B5 4B             LDA  R0,X
315:  C181 F9 4B 00          SBC  R0,Y
316:  C184 95 4B             STA  R0,X
317:  C186 B5 4C             LDA  R0+1,X
318:  C188 F9 4C 00          SBC  R0+1,Y
319:  C18B 95 4C             STA  R0+1,X
320:  C18D 60                RTS
```

Die Routine SUB subtrahiert den Inhalt des Ziel-Registers vom Inhalt des Quell-Registers. Das Ergebnis steht im Quell-Register. Das Programm ist ähnlich aufgebaut wie das Additions-Programm.

Zunächst wird mit der Routine GETDI der Index des Ziel-Registers geholt (310). Dieser Wert steht sowohl im X-Register als auch im Akku. Deshalb kann er gleich vom Akku nach Y gebracht werden (311).

Mit GETSR wird dann der Index des Quell-Registers geholt (312), der nun in X steht.

Vor dem Subtrahieren muß mit SEC das Carry-Bit gesetzt werden (313). Anschließend kann das Low-Byte des Ziel-Registers mit SBC vom Low-Byte des Quell-Registers subtrahiert werden (314-315). Das Ergebnis wird im Quell-Register gespeichert (316). Entsprechend erfolgt die Subtraktion und das Speichern der High-Bytes (317-319).

```
                        ;REGISTER MIT PARAMETER VORBELEGEN
                        ;(SET REGISTER WITH PARAMETER)
324:  C18E 20 CE C0 SETREG   JSR  GETDI    ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
325:  C191 A5 FD             LDA  PAR      ;LOW-BYTE DES PARAMETERS ...
326:  C193 95 4B             STA  R0,X     ;INS REGISTER SPEICHERN
327:  C195 A5 FE             LDA  PAR+1    ;DAS GLEICHE AUCH FUER ...
328:  C197 95 4C             STA  R0+1,X   ;HIGH-BYTE
329:  C199 60                RTS
```

Diese Routine lädt ein Register mit dem angegebenen Parameter.

Als erstes muß die Information über das Ziel-(Destination-) Register geholt werden (324). Dies geschieht in der Routine GETDI (188-192). Der Offset dieses Registers steht in X. Der Wert, mit dem geladen werden soll, muß vorher vom Anwender-Programm in den Speicher PAR gebracht worden sein. Das Low-Byte dieses Parameters wird in den Akku gebracht (325) und in RO,X gespeichert, weil ja in X der Offset des Registers steht (326). Entsprechend wird das High-Byte aus PAR+1 geladen (327) und in RO+1,X gespeichert (328).

Beispiel: SETREG (PAR --> R1)

	Vorher:	Nachher:
Reg.-Adr.	$01	$01
PAR	$1000	$1000
R1	$0000	$1000

```
                ;REGISTER INDIREKT LADEN (LOAD REGISTER INDIRECT)
                ;DIE PARAMETER-ADRESSE STEHT IN PAR
                ;RETURN MIT OPERANDEN IN GREG UND IM GEWUENSCHTEN REGISTER
334:  C19A A5 FD  LRIND  LDA  PAR     ;OPERANDENADRESSE NACH ...
335:  C19C 85 A3         STA  GREG    ;GENERALREGISTER ...
336:  C19E A5 FE         LDA  PAR+1   ;SPEICHERN
337:  C1A0 85 A4         STA  GREG+1
338:  C1A2 20 B7 C0      JSR  LGIND   ;LADE GENERALREGISTER INDIREKT ...
339:  C1A5 4C 45 C1      JMP  SAVG    ;UND SPEICHERE INS GEWUENSCHTE REGISTER
```

Mit dieser Routine wird ein Register indirekt geladen, d.h. mit dem Wert, der in der angegebenen Speicherstelle steht.

Die Nummer des indirekt zu ladenden Registers steht im Ziel-Register (Destination). Dieses Register wird mit dem Inhalt des Speichers geladen, und die Adresse dieses Speichers wird hier als Parameter übergeben.

Diese Adresse also wird aus dem Register PAR geholt und nach GREG gespeichert (334-337). Durch den Aufruf von LGIND (338: 164-172) wird GREG mit dem Inhalt der Adresse, die in GREG steht, geladen.

Um den Wert ins Zielregister zu bringen, benutzen wir die Transfer-Routine TRAREG, allerdings ohne den Aufruf von LOADS (339: 277-282). Dieser Teil von TRAREG wurde SAVREG (= SAve REGister) genannt.

Beispiel: LRIND

	Vorher:	Nachher:
$1000	$01	
$1001	$20	
Reg.-Adr.	$02	$02
PAR	$1000	$1000
R2	$xxxx	$2001

```
                    ;REGISTER INDIREKT SPEICHERN (STORE REGISTER INDIRECT)
                    ;DIE SPEICHERADRESSE STEHT IN PAR
343:  C1A8 20 C5 C0 STOREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
344:  C1AB A0 00            LDY  #$00    ;VORBEREITUNG FUER INDIREKTES LADEN
345:  C1AD B5 4B            LDA  R0,X    ;LOW-BYTE DES REGISTERS HOLEN UND ...
346:  C1AF 91 FD            STA  (PAR),Y ;INDIREKT SPEICHERN
347:  C1B1 C8               INY          ;NAECHSTES BYTE (HIGH-BYTE)
348:  C1B2 B5 4C            LDA  R0+1,X
349:  C1B4 91 FD            STA  (PAR),Y
350:  C1B6 60               RTS
```

Diese Routine besagt: "Speichere den Inhalt eines Registers in der Speicherstelle, deren Adresse in PAR (Parameter) steht." - Beispiel: "Speichere den Inhalt von RO nach $1000." In dem Fall müßte also die Adresse $1000 in PAR stehen.

Nach dem Aufruf von GETSR (343) steht der Index des Quell-Registers in X. Für die mit Y indirekt indizierte Speicherung muß Y zuerst gelöscht werden (344). Der Register-Inhalt wird in den Akku geladen und unter der mit Y indizierten in PAR stehenden Adresse gespeichert (345-346). Nachdem Y inkrementiert wurde (347) wird entsprechend mit dem High-Byte verfahren (348-349).

Beispiel: STOREG

	Vorher:	Nachher:
$1000	$xx	$11
$1001	$xx	$11
Reg.-Adr.	$00	$00
R0	$1111	$1111
PAR	$1000	$1000

6.2 Der 16-Bit-Interpreter

Mit den nun vorliegenden 16-Bit-Routinen besitzen wir die Bausteine, um eine Art "Interpreter" zu bauen, dem wir 16-Bit-Daten übergeben können. Dieser Interpreter soll dann die Daten oder Informationen, die direkt nach dem Aufruf des Interpreters im Anwenderprogramm stehen, einzeln - also Byte für Byte - holen und die Rücksprungadresse unserer 16-Bit-Routinen korrigieren.

Normalerweise wird ja bei einem Unterprogrammaufruf als Rücksprungadresse die Adresse des nächsten Befehls -1 im Stack gespeichert. Da an dieser Stelle im aufrufenden Programm jedoch die zu übergebenden Parameter stehen, muß die Rücksprungadresse so verändert werden, wie es in der Abbildung am Anfang des vorhergehenden Kapitels dargestellt ist: der Rücksprung muß in Abhängigkeit von der Anzahl der Parameter erfolgen. Dies geschieht im Unterprogramm ASSI, dem "ASSembler Interpreter". Der Einfachheit halber sprechen wir im folgenden nur von der Rücksprungadresse, wobei jedoch die Rücksprungadresse -1 gemeint ist.

```
                    ;INTERPRETER FUER 16-BIT-ROUTINEN

36:   C000 20 88 C0 ASSI    JSR  GETADR  ;RUECKSPRUNGADRESSE INS GREG
37:   C003 A0 00            LDY  #$00    ;FUER INDIREKTES LADEN VORBEREITEN
38:   C005 B1 A3            LDA  (GREG),Y ;FUNKTIONSNUMMER HOLEN
39:   C007 29 20            AND  #$20    ;REM 4-BYTE-FUNKTION?
40:   C009 F0 19            BEQ  ASSI1   ;NEIN, NUR 2-BYTE-FUNKTION
```

Mit der Routine GETADR wird zunächst die Rücksprungadresse geholt, die anschließend in GREG steht (36). Diese Adresse ist ein Zeiger auf die gewählte Funktionsnummer.

Um nun diese Funktionsnummer zu holen, müssen wir indirekt indiziert laden, da die indirekte Adressenangabe beim 6510 indiziert sein muß (außer bei JMP). Y wird zunächst auf Null gesetzt (37). Dann wird der Akku mit dem in GREG stehenden Zeiger auf die Funktionsnummer - indirekt indiziert mit Y - geladen (38). Anschließend steht also die Funktionsnummer im Akku.

Da sowohl 2- als auch 4-Byte-Funktionen vorgesehen sind, wird jetzt durch die UND-Verknüpfung mit $20 (dezimal 32) getestet, ob es sich um eine 4-Byte-Funktion handelt (39). Dies geschieht mit dem Befehl

AND

AND ("AND" memory with accumulator), der eine UND-Verknüpfung mit Inhalt des Akkus und dem des angegebenen Speichers durchführt. Das Ergebnis dieser Verknüpfung ist nur dann logisch "wahr" (Ergebnis = 1), wenn beide Operanden den Wert 1 haben. Die folgende Tabelle zeigt die möglichen Kombinationen der Wahrheitswerte bei einer UND-Verknüpfung:

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

Handelt es sich nicht um eine 4-Byte-Funktion, so steht nach dem Vergleich eine Null im Akku, und es wird nach ASSI1 verzweigt, wo die 2-Byte-Funktionen behandelt werden (40). Dieser Test erfolgt mit dem

BEQ

Befehl BEQ (Branch on result EQual zero). Ist der Akku-Inhalt nach diesem Test jedoch ungleich Null, so handelt es sich um eine 4-Byte-Funktion:

```
                        ;4-BYTE-FUNKTION
43:   C00B 20 97 C0           JSR  INCG    ;RUECKSPRUNGADRESSE KORRIGIEREN
44:   C00E 20 97 C0           JSR  INCG    ;PARAMETERADRESSE -3
45:   C011 20 97 C0           JSR  INCG
46:   C014 20 A7 C0           JSR  PUTADR  ;KORRIGIERTE RUECKSPRUNGADRESSE ZURUECKSCHREIBEN
47:   C017 20 9E C0           JSR  DECG    ;PARAMETERADRESSE WIEDERHERSTELLEN
48:   C01A 20 9E C0           JSR  DECG
49:   C01D 20 9E C0           JSR  DECG
50:   C020 A0 03              LDY  #$03    ;4 BYTES UMSPEICHERN (ZAEHLER)
51:   C022 D0 0B              BNE  GETPAR  ;UEBERGABE-PARAMETER UMSPEICHERN
```

In dem Fall müssen wir unsere Rücksprungadresse dadurch korrigieren, daß wir sie um drei Bytes weitersetzen. Dazu wird das Unterprogramm INCG dreimal aufgerufen (43-45). Bei dem in Zeile 40 durchgeführten Test enthielt GREG den Zeiger auf die Funktionsnummer. Da wir hier vier Bytes als Parameter haben, müssen wir also diese dreimalige Erhöhung durchführen, weil die Rücksprungadresse gleich der eigentlichen Adresse minus eins ist.

Nach dieser Korrektur muß die Rücksprungadresse in den Stack zurückgeschrieben werden. Dies geschieht in der Routine PUTADR (46).

Nach dem Aufruf von PUTADR kehren wir zurück zum aufrufenden Programm ASSI. Da wir zuvor dreimal inkrementiert haben (43-45) müssen wir nun dreimal dekrementieren, um wieder zu unserer Funktion zu kommen (47-49).

Anschließend steht also in GREG der Zeiger auf die Funktion und wir müssen nun die vier übergebenen Parameter herausholen: Funktionsnummer, Register, Adresse und Parameter. Dazu laden wir zunächst Y mit $03, um von drei bis Null herunterzählen zu können (50). Fällt der Test auf Null (51) positiv aus, so wird zur Routine GETPAR verzweigt, also der folgende Programmteil ASSI1 übersprungen. Normalerweise müßte hier zwar JMP stehen, da wir jedoch wissen, daß der Wert von Y ungleich Null ist, kann der Befehl BNE verwendet werden, wodurch wir gegenüber dem 3-Byte-Befehl JMP ein Byte sparen.

Nachdem (je nach Funktionstyp) 2 oder 4 Parameter geholt worden sind, wird das Programm in beiden Fällen (bei der Markierung GETPAR) gleich fortgesetzt. Bevor wir mit der Beschreibung an dieser Stelle fortfahren, besprechen wir aber noch den Programmteil zur Korrektur der Rücksprungadresse bei einer 2-Byte-Funktion:

```
                        ;2-BYTE-FUNKTION
54:   C024 20 97 C0 ASSI1   JSR  INCG    ;RUECKSPRUNGADRESSE KORRIGIEREN
55:   C027 20 A7 C0         JSR  PUTADR  ;RUECKSPRUNGADRESSE ZURUECKSPEICHERN
56:   C02A 20 9E C0         JSR  DECG    ;PARAMETERADRESSE WIEDERHERSTELLEN
57:   C02D A0 01            LDY  #$01    ;2 BYTES UMSPEICHERN (ZAEHLER)
```

Dieser Programmteil wird dann durchlaufen, wenn eine 2-Byte-Funktion vorliegt: in dem Fall werden die Zeilen 43-51 übersprungen, da zum Sichern der Adresse nur einmal inkrementiert werden muß (54) und nach dem Sichern (55) demnach auch nur einmal dekrementiert werden muß, um zur Anfangsadresse der Funktion zu gelangen (56). Y wird bei einer 2-Byte-Funktion mit dem Wert $01 geladen, da ja nur zwei Bytes geholt werden müssen (57).

In dem mit GETPAR gekennzeichneten Programmteil werden nun die Parameter geholt:

```
                    ;PARAMETER UMSPEICHERN
59:   C02F
                    ;FUNKTIONSNUMMER IN FNC
                    ;REGISTERINDEX IN   REGADR (HIGH-NIBBLE FUER QUELLE)
                    ;PARAMETER IN       PAR    (2 BYTES)
63:   C02F B1 A3  GETPAR  LDA  (GREG),Y
64:   C031 99 FB 00       STA  FNC,Y
65:   C034 88              DEY
66:   C035 10 F8           BPL  GETPAR
```

Wir indizieren indirekt mit Y (63), wobei Y gleich eins ist bei 2-Byte-Funktionen und gleich drei bei 4-Byte-Funktionen. Dieser Wert wird direkt indiziert nach FNC,Y gespeichert. Je nach Funktion werden zwei oder vier Bytes geholt und gespeichert, indem Y jedesmal mit dem Befehl DEY (DEcrement index Y) um eins vermindert wird (65).

DEY

Da der Zähler Y in der Schleife heruntergezählt wird, werden die vier zu einer Funktion gehörenden Bytes FNC, REGADR, PAR1, PAR2 von hinten an gelesen. Der letzte Wert ist also FNC, die Funktionsnummer. Im nächsten Programmteil, in dem die der Funktion zugeordnete Routine aufgerufen wird, wird davon Gebrauch gemacht, daß dieser Wert auch noch im Akku steht:

```
                    ;SPRUNG ZUR GEWAEHLTEN FUNKTION DURCH JSR
                    ;ZUERST DIE FUNKTIONS-ANFANGSADRESSE -1 IN DEN STACK SCHIEBEN,
                    ;DANN JUMP TO SUBROUTINE

72:   C037 0A             ASL          ;#2 = OFFSET IN TABELLE
73:   C038 A8             TAY          ;OFFSET FUER INDIZIERTES LADEN
74:   C039 B9 41 C0       LDA  FNCTAB-1,Y  ;LOW-BYTE ...
75:   C03C 48             PHA          ;IN DEN STACK
76:   C03D B9 40 C0       LDA  FNCTAB-2,Y  ;HIGH-BYTE ...
77:   C040 48             PHA          ;AUCH
78:   C041 60       DOIT  RTS          ;VERZWEIGUNG ZUR GEWAEHLTEN FUNKTION
```

Der Akku-Inhalt wird um eine Bitstelle nach links verschoben, also verdoppelt (72). Dabei erhält Bit 0 den Wert 0 und Bit 7 wird in das Übertragsbit C geschoben. Dieser Wert wird ins Y-Register kopiert (73), um als Index für die Funktionstabelle zu dienen.

Die Adressen der einzelnen Funktions-Routinen sind in einer Tabelle gespeichert (83-117). Low- und High-Byte der Anfangsadresse einer Funktion werden in den Akku geladen und in den Stack geschoben (74-77).

Eine Erweiterung dieses "Programmpaketes" ist leicht möglich, indem einfach die Anfangsadresse der neuen Routine in die Tabelle geschrieben wird.

7 | Grafik-Anwendungen der Assembler-Routinen

7. Grafik-Anwendungen der Assembler-Routinen

In diesem Kapitel werden Routinen für die hochauflösende Grafik sowie die Sprite-Programmierung vorgestellt. Dabei werden die bereits entwickelten 16-Bit-Routinen verwendet.

7.1 16-Bit-Routinen zur hochauflösenden Grafik

7.1.1 Grafik einschalten: GON

```
                        ;GRAFIK EINSCHALTEN
540:   C304 AD 11 D0 GON      LDA  BITMAP
541:   C307 8D 28 C3          STA  VIDEO1
542:   C30A A9 3B             LDA  #$3B
543:   C30C 8D 11 D0          STA  BITMAP
544:   C30F AD 18 D0          LDA  SCRMEM
545:   C312 8D 29 C3          STA  VIDEO2
546:   C315 A9 18             LDA  #$18
547:   C317 8D 18 D0          STA  SCRMEM
548:   C31A 60                RTS
```

Mit der Routine GON wird die hochauflösende Grafik eingeschaltet.

Der Inhalt des Video-Chip-Kontroll-Registers BITMAP ($DO11) wird in den im Programm vereinbarten Speicher VIDEO1 ($C328) gebracht (540-541), damit er beim Ausschalten der Grafik wieder auf den ursprünglichen Wert gesetzt werden kann. Darauf wird mit dem Wert $3B eine Bit-Konfiguration nach BITMAP gespeichert, durch die der Grafik-Modus eingeschaltet wird (542-543).

Dazu muß noch ein bestimmtes Bitmuster in das Video-Chip-Memory-Control-Register SCRMEM ($DO18) gespeichert werden (546-547), dessen urspünglicher Inhalt nach VIDEO2 ($C329) gebracht wird.

7.1.2 Grafik ausschalten: GOFF

```
                    ;GRAFIK AUSSCHALTEN
551:  C31B AD 28 C3 GOFF    LDA  VIDEO1
552:  C31E 8D 11 D0         STA  BITMAP
553:  C321 AD 29 C3         LDA  VIDEO2
554:  C324 8D 18 D0         STA  SCRMEM
555:  C327 60               RTS

557:  C328 00       VIDEO1  .BYTE$00
558:  C329 00       VIDEO2  .BYTE$00
```

Mit der Routine GOFF wird die hochauflösende Grafik ausgeschaltet.

Die vor dem Einschalten der Grafik "geretteten" ursprünglichen Werte von BITMAP und SCRMEM werden wieder zurückgespeichert, um den Grafik-Modus auszuschalten.

7.1.3 Grafikspeicher löschen: GCLR

```
                    ;GRAFIKSPEICHER LOESCHEN ($2000-$3FFF)
561:  C32A 20 00 C0 GCLR    JSR  ASSI
562:  C32D 20 00 00         .BYTE$20,$00,$00,$20  ;R0=$2000
563:  C331 20 00 C0         JSR  ASSI
564:  C334 20 01 FF         .BYTE$20,$01,$FF,$3F  ;R1=$3FFF
565:  C338 A9 00            LDA  #$00
566:  C33A 4C B7 C1         JMP  FILL
```

Mit der Routine GCLR wird der Grafikspeicher gelöscht.

Durch den Aufruf der Routine Nr. 32 ($20) wird die Funktion SETREG gewählt, um R0 auf den Anfangswert des Grafikspeichers ($2000) zu setzen (561-562). Mit derselben Routine wird R1 der Endwert des Grafikspeichers ($3FFF) zugewiesen (563-564).

Nachdem der Akku mit $00 geladen wurde (565), wird zum Unterprogramm FILL verzweigt (566), das den Speicherbereich von R0 bis R1 mit dem im Akku befindlichen Wert "füllt", d.h. das angegebene Zeichen wird in jede Speicherzelle dieses Bereiches geschrieben:

```
                    ;FILL MEMORY
                    ;STARTADRESSE = R0
                    ;ENDADRESSE   = R1
                    ;DATA IN ACCUMULATOR
357:  C1B7 A0 00  FILL    LDY  #$00    ;INDIREKTES SPEICHERN VORBEREITEN
358:  C1B9 91 4B  FL1     STA  (R0),Y
359:  C1BB E6 4B          INC  R0
360:  C1BD D0 02          BNE  TSTEND
361:  C1BF E6 4C          INC  R0+1
362:  C1C1 A6 4C  TSTEND  LDX  R0+1
363:  C1C3 E4 4E          CPX  R1+1
364:  C1C5 D0 F2          BNE  FL1
365:  C1C7 A6 4B          LDX  R0
366:  C1C9 E4 4D          CPX  R1
367:  C1CB 90 EC          BCC  FL1
368:  C1CD 60             RTS
```

Diese Routine benutzt die Register R0 und R1, wobei R0 die Startadrese und R1 die Endadresse des gewünschten Speicherbereiches enthält.

Beim 6510 kann man in einer Schleife normalerweise nur 256 Bytes bearbeiten. Mit diesem Unterprogramm entfällt diese Einschränkung, und die Bytezahl ist nur durch den verfügbaren Speicherbereich begrenzt.

Das zur Indizierung verwendete Y-Register wird auf den Anfangswert 0 gesetzt (357). Dann kann der erste Speicher mit dem im Akku stehenden Wert gefüllt werden (358).

Das Low-Byte der Anfangsadresse (R0) wird inkrementiert (359). Sofern es anschließend ungleich Null ist, wird nach TSTEND verzweigt (360), wo überprüft wird, ob das High-Byte von R0 bereits gleich dem in R1 befindlichen Endwert ist (362-363).

War der Inhalt von R0 vor dem Inkrementieren gleich 255, so ist er danach gleich 0, so daß auch das High-Byte von R0 (also R0+1) inkrementiert werden muß, bevor der folgende Test stattfindet (361).

Sind die beiden High-Bytes von R0 und R1 noch nicht gleich (364), wird der nächste Speicher mit dem im Akku befindlichen Wert gefüllt (358).

Sind jedoch die beiden High-Bytes der Anfangs- und Endadresse gleich, so werden die Low-Bytes von R0 und R1 miteinander verglichen: solange sie ungleich sind, wird wieder nach FL1 verzweigt, ansonsten ist das Programm beendet, da der Anfangswert nun mit dem Endwert übereinstimmt (365-367).

7.1.4 Hintergrundfarbe setzen: GCOL

```
                    ;HINTERGRUNDFARBE SETZEN ($400-$7E7)
                    ;FARBE IM AKKU
570:  C33D 48       GCOL    PHA
571:  C33E 20 00 C0         JSR  ASSI
572:  C341 20 00 00         .BYTE$20,$00,$00,$04  ;R0=$400
573:  C345 20 00 C0         JSR  ASSI
574:  C348 20 01 E7         .BYTE$20,$01,$E7,$07  ;R1=$7E7
575:  C34C 68               PLA
576:  C34D 4C B7 C1         JMP  FILL
```

Mit der Routine GCOL wird die Hintergrundfarbe gesetzt.

Da bei der Grafik-Darstellung der normale Bildschirmspeicher als Farbspeicher dient, wird hier die Routine FILL mit der Anfangsadresse $400 und der Endadresse $7E7 aufgerufen. Die Nummer der Farbe muß vorher in den Akku gebracht werden.

7.1.5 Adressberechnung für Grafikspeicher: GRAFAD

```
                    ;ADRESSE EINES GRAPHIKSPEICHERS BERECHNEN
                    ; X-KOORD. IN R0
                    ; Y-KOORD. IN R1
496:  C2AE A5 4D    GRAFAD  LDA  R1      ;Y-KOORD., 1 BYTE
497:  C2B0 4A               LSR
498:  C2B1 4A               LSR
499:  C2B2 4A               LSR          ;ZEILENNUMMER
500:  C2B3 A8               TAY
```

```
501:  C2B4 B9 E7 C2          LDA  HIGTAB,Y
502:  C2B7 18                CLC
503:  C2B8 65 4C             ADC  R0+1     ;X-KOORD. (HIGH-BYTE) ALS KORREKTUR
504:  C2BA 85 5C             STA  RSAV+1   ;SICHERN IN RSAV (HIGH-BYTE)
505:  C2BC 98                TYA
506:  C2BD 29 03             AND  #$03     ;ZEILENNUMMER MOD 4
507:  C2BF A8                TAY
508:  C2C0 A5 4B             LDA  R0
509:  C2C2 29 F8             AND  #$F8     ;INT(X-KOORD./8)*8
510:  C2C4 79 00 C3          ADC  LOWTAB,Y
511:  C2C7 85 5B             STA  RSAV
512:  C2C9 90 03             BCC  GRAF1
513:  C2CB E6 5C             INC  RSAV+1
514:  C2CD 18                CLC
515:  C2CE A5 4D    GRAF1    LDA  R1       ;Y-KOORD.
516:  C2D0 29 07             AND  #$07
517:  C2D2 65 5B             ADC  RSAV
518:  C2D4 85 5B             STA  RSAV

520:  C2D6 A5 4B    GETBIT   LDA  R0
521:  C2D8 29 07             AND  #$07
522:  C2DA A8                TAY
523:  C2DB A9 80             LDA  #$80
524:  C2DD C0 00             CPY  #$00
525:  C2DF F0 05             BEQ  GBOUT
526:  C2E1 18        GBLOOP  CLC
527:  C2E2 6A                ROR
528:  C2E3 88                DEY
529:  C2E4 D0 FB             BNE  GBLOOP
530:  C2E6 60        GBOUT   RTS

532:  C2E7 20 21 22 HIGTAB   .BYTE$20,$21,$22,$23,$25
533:  C2EC 26 27 28          .BYTE$26,$27,$28,$2A,$2B
534:  C2F1 2C 2D 2F          .BYTE$2C,$2D,$2F,$30,$31
535:  C2F6 32 34 35          .BYTE$32,$34,$35,$36,$37
536:  C2FB 39 3A 3B          .BYTE$39,$3A,$3B,$3C,$3E
537:  C300 00 40 80 LOWTAB   .BYTE$00,$40,$80,$C0
```

Anhand der in R0 bzw. in R1 gespeicherten X- bzw. Y-Koordinaten wird die Adresse eines Grafikspeichers ermittelt. Die Adresse steht anschließend in RSAV und RSAV+1 und die Bit-Position innerhalb dieses Bytes im Akku zur Verfügung. Das Programm GRAFAD wird von SETBIT (478-482) und CLRBIT (485-490) aufgerufen.

Die Berechnung erfolgt nicht rein arithmetisch, sondern mit Hilfe einer Tabelle (532-537), wodurch sich eine beträchtliche Zeitersparnis ergibt.

Um den Aufbau dieser Tabelle zu verstehen, müssen Sie wissen, daß der Bildschirm in der Textdarstellung in 25 Zeilen zu jeweils 40 Zeichen aufgeteilt ist. Jedes Zeichen wiederum ist in einem Raster von 8x8=64 Punkten dargestellt, d.h. durch 8 Bytes. Bei der Darstellung der hochauflösenden Grafik ist es nun möglich, jeden einzelnen dieser Bildschirmpunkte gezielt ein- oder auszuschalten. Insgesamt ergibt sich auf diese Weise eine Auflösung von 320 (=40*8) mal 200 (=25*8), also 64000 Punkten, für die 8000 Bytes an Speicherplatz benötigt werden.

Die Anfangsadresse des Grafik-Speichers liegt bei $2000 (8192 dezimal). Da in jeder Zeile (X-Richtung) 320 Punkte dargestellt werden können, ist z.B. die Adresse des ersten Speicherplatzes der zweiten Zeile: 8192+320 = 8512 bzw. $2140 usw.

$2000	$2008	$2010	$2018	.....	$2138
$2001	$2009	.	.		$2139
$2002	$200A	.	.		$213A
$2003	$200B	.	.		$213B
$2004	$200C	.	.		$213C
$2005	$200D	.	.		$213D
$2006	$200E	.	.		$213E
$2007	$200F	.	.		$213F
$2140	$2048	$2150	$2158	.....	
$2141	$2049	.	.		
$2142	$214A	.	.		
$2143	$214B	.	.		
$2144	$214C	.	.		
$2145	$214D	.	.		
$2146	$214E	.	.		
$2147	$214F	.	.		
.....					
.....					
.....					

Wie Sie nun leicht überprüfen können, enthält unsere Tabelle zunächst einmal die High-Bytes der jeweils ersten Adresse der 25 Bildschirmzeilen: $20, $21, $22 usw. Daran schließen sich vier Low-Byte-Werte an: $00, $40, $80, $C0 (dezimal: 0, 64, 128 und 192).

Es stellt sich die Frage, warum die High-Bytes der 25 Zeilenanfänge jeweils explizit in der Tabelle angegeben sind, für die Low-Bytes jedoch nur 4 Werte? Dies liegt daran, daß sich diese Werte in der angegebenen Reihenfolge immer wiederholen:

```
Zeile 0:  $2000
Zeile 1:  $2140
Zeile 2:  $2280
Zeile 3:  $23C0

Zeile 4:  $2500
Zeile 5:  $2640
Zeile 6:  $2780
Zeile 7:  $28C0      usw.
```

Mit diesen Kenntnissen können wir mit der Programmbeschreibung beginnen.

Um das High-Byte der Adresse des Grafik-Speichers zu berechnen, wird zunächst die Y-Koordinate bearbeitet, deren Wert zuvor in den Speicher R1 gebracht werden muß. Da die Y-Koordinate nur Werte von 0 bis 199 annehmen kann, wird nur ein Byte benötigt. Der Y-Wert wird in den Akku geladen (496) und durch dreimaliges Verschieben nach rechts durch 8 geteilt (497-499). Auf diese Weise erhält man die Textzeilennummer, die von 0 bis 24 reichen kann.

Diese Zeilennummer wird nun ins Y-Register gebracht (500) und als Index verwendet, um das High-Byte des Zeilenanfangs aus der Tabelle zu ermitteln, das in den Akku geladen wird (501).

Dieser Wert muß jetzt eventuell noch korrigiert werden, da in einer Zeile mehr als 255 Punkte gesetzt werden können. Diese Korrektur erfolgt dadurch, daß das High-Byte der X-Koordinate zur Y-Koordinate hinzuaddiert wird (502-503). Das Ergebnis (also das High-Byte) wird in RSAV+1 gespeichert (504). Da hier kein Überlauf auftreten kann, muß auch keine entsprechende

Überprüfung stattfinden: das Ergebnis kann höchstens $3E+$01 = $3F (dezimal 63) betragen, da das High-Byte von X nur 0 oder 1 sein kann.

Bisher haben wir die Y-Koordinate etwas "grob" betrachtet, nämlich in den Einheiten einer Textzeile. Tatsächlich müssen wir aber berücksichtigen, daß jedes "Textkästchen" aus 8 Bytes besteht. Dies geschieht in den folgenden Programmteilen.

Als nächstes wird das Low-Byte der Adresse des Grafik-Speichers berechnet. Da die Nummer der Textzeile noch im Index-Register Y steht, kann sie von da aus direkt in den Akku gebracht werden (505). Durch die UND-Verknüpfung mit $03 wird eine Modulo 4-Berechnung durchgeführt, wodurch sich Werte von 0 bis 3 ergeben (506). Das Ergebnis wird wieder ins Y-Register gebracht (507) und dient als Zeiger auf das zugehörige Low-Byte.

Um die gewünschte Adresse zu erhalten, muß man berücksichtigen, daß man (in Richtung der X-Koordinate) in die jeweils nächste Textspalte gelangt, indem man eine Adresse um 8 erhöht. Um innerhalb eines "Textkästchens" das nächste Byte in Y-Richtung zu bestimmen, muß zu der Adresse der Wert 1 hinzuaddiert werden.

Deshalb wird jetzt der Wert der X-Koordinate (aus R0) in den Akku geladen (508) und mit $F8 UND-verknüpft, d.h. nur die drei unteren Bits bleiben gesetzt (509). Dies entspricht der folgenden Berechnung: INT(X/8)*8, oder anders gesagt: das Low-Byte der X-Koordinate wird auf das nächste Vielfache von 8 abgerundet. Dazu wird dann das durch das Y-Register bestimmte Low-Byte aus der Tabelle hinzuaddiert (510). Das Ergebnis, also das vorläufige Low-Byte, wird in RSAV gespeichert (511).

Entstand bei dieser Addition kein Übertrag, so wird nach GRAF1 (515) verzweigt, ansonsten wird das High-Byte der Adresse um 1 erhöht (513) und das Carry-Bit für die nächste Addition wieder gelöscht (514).

Um auf den endgültigen Wert für das Low-Byte zu kommen, muß der Wert der Y-Koordinate nun noch um eine Modulo 8-Addition erhöht werden. Dies geschieht durch die UND-Verknüpfung mit 7, wodurch sich Werte von 0 bis 7 ergeben. Das Ergebnis wird wieder in RSAV gespeichert (515-518).

In dem sich anschließenden Programmteil GETBIT (520-530) wird jetzt die Position des gewünschten Bits innerhalb eines Bytes berechnet. Dabei muß berücksichtigt werden, daß die Grafikpunkte von links nach rechts durchnummeriert sind - also genau entgegengesetzt zur Nummerierung der Bits innerhalb eines Bytes.

Die X-Koordinate wird in den Akku gebracht (520). Durch die UND-Verknüpfung mit 7 wird eine Modulo 8-Berechnung durchgeführt (521). Das Ergebnis - also ein Wert von 0 bis 7 - wird ins Y-Register gebracht (522) und dient als Zähler für die folgende Schleife, die entsprechend oft durchlaufen wird. Vor Beginn dieser Schleife wird der Akku mit $80 initialisiert (binär: 10000000), so daß also nur das höchstwertige Bit gesetzt ist (523), und es wird abgefragt, ob Y gleich 0 ist: in dem Fall brauchte die Schleife überhaupt nicht durchlaufen zu werden, und die Adressberechnung wäre beendet (524-525). Ist dies nicht der Fall, wird die Schleife durchlaufen: der mit $80 vorbesetzte Inhalt des Akkus wird (nach dem Löschen des Carry-Bits) um ein Bit nach rechts rotiert, so daß jetzt also das zweite Bit (von links) für den Grafikpunkt gesetzt ist. Y wird dekrementiert, und sofern es noch nicht gleich 0 ist, wird die Schleife noch einmal durchlaufen (526-529). Anschließend steht somit die Bit-Position im Akku zur Verfügung.

7.1.6 Grafikpunkt setzen/löschen: SETBIT/CLRBIT

```
474:  C293 2C D2 C1 TOGGLE   BIT  COLOR
475:  C296 30 0A            BMI  CLRBIT

                     ;SET GRAPHIC DOT
478:  C298 20 AE C2 SETBIT   JSR  GRAFAD
479:  C29B A0 00    DOBIT    LDY  #$00
480:  C29D 11 5B             ORA  (RSAV),Y
481:  C29F 91 5B             STA  (RSAV),Y
482:  C2A1 60                RTS

                     ;CLEAR GRAPHIC DOT
485:  C2A2 20 AE C2 CLRBIT   JSR  GRAFAD
486:  C2A5 49 FF             EOR  #$FF
487:  C2A7 A0 00             LDY  #$00
488:  C2A9 31 5B             AND  (RSAV),Y
489:  C2AB 91 5B             STA  (RSAV),Y
490:  C2AD 60                RTS
```

Die Routinen SETBIT bzw. CLRBIT dienen zum Setzen bzw. Löschen eines Grafikpunktes. Es wird vorausgesetzt, daß die Koordinaten dieses Punktes in R0 (X-Koordinate) und R1 (Y-Koordinate) stehen. Beide Programme verwenden das soeben besprochene Programm GRAFAD zur Berechnung der Adresse eines Grafikspeichers.

Vor dem Setzen bzw. Löschen eines Grafik-Punktes wird von dem gleich folgenden Programm zum Zeichnen einer Linie die kurze Routine TOGGLE (474-475) durchlaufen:

Hier wird das COLOR-Byte untersucht, um festzustellen, ob ein Punkt gesetzt oder gelöscht werden soll. Im gleich folgenden Programm-Beispiel zum Zeichnen einer Linie wird das COLOR-Byte auf $00 gesetzt, wenn ein Punkt gesetzt werden soll und auf $FF, wenn ein Punkt gelöscht werden soll. In der Routine TOGGLE wird davon Gebrauch gemacht, daß $FF im Zweierkomplement eine negative Zahl ist. An dieser Stelle könnten weitere Abfragen eingebaut werden, um verschiedene Farben benutzen zu können.

BIT

Durch den Befehl BIT (test BITs in memory with accumulator) wird der Inhalt des Akkumulators mit dem Inhalt der angegebenen Speicherstelle (hier also mit dem COLOR-Byte) UND-verknüpft. Ist das Ergebnis der Verknüpfung gleich Null, so wird das Z-Flag gesetzt: Z = 1, sonst ist Z = 0. Das Ergebnis selbst wird jedoch nicht gespeichert, und der Akkumulatorinhalt bleibt unverändert. Außerdem werden die Bits 6 und 7 der angegebenen Speicherstelle in die Flags V bzw. N des Statusregisters übertragen. Dies wird bei der anschließenden Abfrage ausgenutzt.

BMI

Mit dem folgenden Verzweigungsbefehl BMI (Branch on result MInus) erfolgt - in Abhängigkeit vom COLOR-Byte - die Verzweigung zu SETBIT oder CLRBIT. Die Verzweigung zu der angegebenen Adresse (hier also CLRBIT) erfolgt dann, wenn das Vorzeichenflag N = 1 ist. Wie bei den übrigen Verzweigungsbefehlen muß die Adresse relativ im Zweierkomplement angegeben werden.

Grafikpunkt setzen: SETBIT

Zunächst wird das Programm GRAFAD aufgerufen, so daß anschließend die Adresse des Grafikspeichers in RSAV und RSAV+1 und die Bit-Position im Akku steht (478).

Mit SETBIT wird nun einfach der alte Inhalt dieser Adresse ODER-verknüpft mit der Bit-Position. Da die Adressierung indiziert erfolgen muß, wird das Index-Register Y zuvor mit 0 geladen (479). Durch die anschließende ODER-Verknüpfung von RSAV mit dem Akku (der die Bit-Position enthält) wird das gewünschte Bit dieses Speichers gesetzt und wieder unter derselben Adresse gespeichert (480-481).

Grafikpunkt löschen: CLRBIT

Diese Routine stellt das Gegenstück zu SETBIT dar. Auch hier wird die Adressberechnung des Grafikspeichers durch den Aufruf von GRAFAD durchgeführt (485).

Die Bit-Position wird durch das exklusive ODER mit $FF verknüpft (486). Dies geschieht mit dem Befehl EOR ("Exclusive OR" memory with accumulator). Das Ergebnis ist nur dann 0, wenn beide Operanden den Wert 1 haben. In der folgenden Tabelle sind die Ergebnisse der Verknüpfung zweier Operanden mit dem ausschließenden logischen ODER dargestellt:

EOR

A	B	A EOR B
1	1	0
1	0	1
0	1	1
0	0	0

Durch die Verknüpfung mit EOR wird die Bit-Position mit 0 gekennzeichnet. Durch die UND-Verknüpfung des Akkus mit dem Low-Byte der Speicheradresse (RSAV) wird das betreffende Bit dann gezielt gelöscht und wieder in RSAV gespeichert (488-489).

7.1.7 Zeichnen und Löschen von Geraden: PLOT/ERASE LINE

```
                ;LINIE ZEICHNEN/LOESCHEN (PLOT LINE/ERASE LINE)

                ;STARTKOORDINATE (XS,YS)  R0,R1
                ;ENDKOORDINATE   (XE,YE)  R2,R3

                ;RSAV WIRD ALS ADRESSPEICHER BENUTZT (BASIS)

378:  C1CE A9 00    PLOT     LDA  #$00
379:  C1D0 F0 03             BEQ  ERASE1
380:  C1D2 00       COLOR    .BYTE$00
381:  C1D3 A9 FF    ERASE    LDA  #$FF
382:  C1D5 8D D2 C1 ERASE1   STA  COLOR
```

Die Programme zum Zeichnen (PLOT LINE) bzw. Löschen (ERASE LINE) einer Linie unterscheiden sich nur durch das Flag COLOR (380), das bestimmt, ob gezeichnet oder gelöscht werden soll: ist der Wert dieses Bytes gleich $00, so wird eine Linie gezeichnet, ist er $FF, so wird eine Linie gelöscht. Die Flag-Bezeichnung COLOR wurde gewählt, weil es grundsätzlich auch möglich ist, durch einen entsprechenden Wert die Farbe der zu zeichnenden Linie zu bestimmen.

Je nachdem, ob eine Linie gezeichnet oder gelöscht werden soll, wird der Akku mit $00 (378) bzw. $FF (381) geladen. Dieser Inhalt des Akkus wird dann in das als Flag verwendete Byte COLOR gebracht (382).

Das ganze folgende Programm ist für PLOT LINE und ERASE LINE identisch und unterscheidet sich - wie gesagt - nur durch den Wert des COLOR-Flags.

Die Start-Koordinaten (XS,YS) müssen in den Registern R0 (XS) bzw. R1 (YS) stehen, die End-Koordinaten (XE,YE) in den Registern R2 (XE) bzw. R3 (YE).

Zunächst werden die X- und Y-Koordinaten-Anteile Delta-X und Delta-Y der Linie berechnet, indem die Differenzen der Anfangs- und End-Koordinaten gebildet werden:

```
                    ;DELTA X,Y BILDEN
384:  C1D8 20 93 C2         JSR  TOGGLE  ;1. PUNKT
385:  C1DB 20 00 C0         JSR  ASSI
386:  C1DE 0E 20            .BYTE$0E,$20  ;DELTAX -> R2
387:  C1E0 20 00 C0         JSR  ASSI
388:  C1E3 0E 31            .BYTE$0E,$31  ;DELTAY -> R3
```

Dazu wird zunächst der erste Grafik-Punkt gesetzt (384). Dann wird mit der Funktion Nr. 14 ($0E, Subtraktion) die X-End-Koordinate (R2) von der X-Anfangs-Koordinate (R0) subtrahiert. Das Ergebnis, Delta-X, steht anschließend in R2 zur Verfügung (385-386).

Zur Berechnung von Delta-Y wird entsprechend die Differenz der Y-End-Koordinate (R3) von der Y-Anfangs-Koordinate (R1) gebildet, die danach in R3 gespeichert ist (387-388).

Auf diese Weise ist nun die Richtung der Geraden ermittelt worden.

In Abhängigkeit von der Richtung der Geraden wird das Inkrement bestimmt: ist die Anfangs-Koordinate kleiner als die End-Koordinate, so ist das Inkrement positiv (+1), im umgekehrten Falle ist es negativ (-1).

Das Inkrement für X wird in den Zeilen 390-399 bestimmt:

```
                    ;INKREMENT FUER X BESTIMMEN
390:  C1E5 A5 50    GIFX    LDA  R2+1    ;DELTAX POS ODER NEG ?
391:  C1E7 10 0E            BPL  XPOS    ;DELTAX IST POSITIV
392:  C1E9 A9 FF            LDA  #$FF
393:  C1EB 85 53            STA  R4
394:  C1ED 85 54            STA  R4+1    ;INKREMENT FUER X IST -1
395:  C1EF 20 00 C0         JSR  ASSI
396:  C1F2 07 20            .BYTE$07,$20  ;VORZEICHEN-WECHSEL
397:  C1F4 4C FE C1         JMP  GIFY
398:  C1F7 20 00 C0 XPOS    JSR  ASSI
399:  C1FA 20 04 01         .BYTE$20,$04,$01,$00  ;INKREMENT FUER X IST +1
```

Mit dem Befehl BPL wird überprüft, ob Delta-X positiv ist (391). Dazu muß nur das High-Byte von Delta-X untersucht werden (390). Ist Delta-X positiv, so wird zum Label XPOS (398) verzweigt, da der Inkrementwert für X +1 betragen muß. In dem Fall wird mit der Funktion Nr. 32 ($20, SET REGISTER) dem Register R4 der Wert 1 zugewiesen (398-399).

Ist Delta-X jedoch negativ, so wird R4 wird mit $FF (also -1 im Zweier-Komplement) geladen (392-394) und es wird der Absolutwert von Delta-X ermittelt, d.h. der Wert wird durch Änderung des Vorzeichens (Funktion Nr. 7) positiv gemacht (395-396). Danach wird zur Inkrement-Bestimmung für Y verzweigt (397).

Die Inkrement-Bestimmung für Y geschieht entsprechend (401-410):

```
                    ;INKREMENT FUER Y BESTIMMEN
401:  C1FE A5 52    GIFY    LDA  R3+1    ; DELTAY POS ODER NEG ?
402:  C200 10 0E            BPL  YPOS    ;DELTAY IST POSITIV
403:  C202 A9 FF            LDA  #$FF
404:  C204 85 55            STA  R5
405:  C206 85 56            STA  R5+1    ;INKREMENT FUER Y IST -1
406:  C208 20 00 C0         JSR  ASSI
407:  C20B 07 30            .BYTE$07,$30  ;VORZEICHEN-WECHSEL
408:  C20D 4C 17 C2         JMP  DRAW
409:  C210 20 00 C0 YPOS    JSR  ASSI
410:  C213 20 05 01         .BYTE$20,$05,$01,$00  ;INKREMENT FUER Y IST +1
```

Vor dem Zeichnen wird untersucht, ob der X- oder der Y-Anteil größer ist. Das Zeichnen wird durch die längere Strecke bestimmt, d.h. der jeweilige Grafik-Punkt wird in Richtung des größeren Koordinaten-Anteils gesetzt. Dies geschieht in der Routine DRAW (412-415).

BCS Der Programmablauf wird durch den Befehl BCS (Branch on Carry Set) gesteuert, der eine Verzweigung zu der angegebenen Adresse plus dem angegebenen Abstand (-128 bis +127) bewirkt, sofern das Carry-Bit gleich 1 ist.

```
412:  C217 20 00 C0 DRAW    JSR  ASSI    ; DELTAX > DELTAY ?
413:  C21A 0C 23            .BYTE$0C,$23  ;VERGLEICHE R2 MIT R3
414:  C21C B0 03            BCS  DRAWX   ; DELTAX IST GROESSER, ZEICHNEN NACH X-ACHSE
415:  C21E 4C 5E C2         JMP  DRAWY
```

Falls Delta-X größer ist als Delta-Y, so wird nach DRAWX (417) verzweigt, im anderen Fall nach DRAWY (447).

```
417:  C221 A5 4F    DRAWX   LDA  R2      ;IST DELTAX = 0 ?
418:  C223 D0 04            BNE  DRX1
419:  C225 A5 50            LDA  R2+1
420:  C227 F0 69            BEQ  DRFINI  ;WENN JA, FERTIG

422:  C229 20 00 C0 DRX1    JSR  ASSI    ;KORREKTUR-REGISTER IST R6
423:  C22C 0B 26            .BYTE$0B,$26  ;DELTAX -> R6
424:  C22E 20 00 C0         JSR  ASSI
425:  C231 05 60            .BYTE$05,$60  ;R6/2 -> KORREKTUR
426:  C233 20 00 C0         JSR  ASSI
427:  C236 0B 27            .BYTE$0B,$27  ;R7 = ZAEHLER

429:  C238 20 00 C0 DRX2    JSR  ASSI
430:  C23B 0D 63            .BYTE$0D,$63  ;KORR+DY->KORR
431:  C23D 20 00 C0         JSR  ASSI
432:  C240 0C 62            .BYTE$0C,$62  ;VERGLEICHE KORR,DX
433:  C242 90 0A            BCC  DRX3    ;KLEINER, DY BLEIBT
                       ; KORR>DX, BEIDE KOORDINATEN ANDERN SICH
435:  C244 20 00 C0         JSR  ASSI
436:  C247 0E 62            .BYTE$0E,$62 ;KORR-DX->KORR
437:  C249 20 00 C0         JSR  ASSI
438:  C24C 0D 15            .BYTE$0D,$15  ;AENDERT Y-KOORD.
439:  C24E 20 00 C0 DRX3    JSR  ASSI
440:  C251 0D 04            .BYTE$0D,$04  ;AENDERT X-KOORD.
441:  C253 20 93 C2         JSR  TOGGLE  ;SET OR CLEAR BIT
442:  C256 20 00 C0         JSR  ASSI
443:  C259 04 70            .BYTE$04,$70  ;WEG DEKREMENTIEREN
444:  C25B D0 DB            BNE  DRX2
445:  C25D 60               RTS
```

Zunächst wird abgefragt, ob Delta-X gleich Null ist (417-420). Ist dies der Fall, so ist die Routine beendet, und es erfolgt der Rücksprung zum aufrufenden Programm. Diese Abfrage entpricht einer DO ... WHILE Schleife: solange die Bedingung erfüllt ist (also Delta-X ungleich Null ist) werden die nachfolgenden Programmschritte ausgeführt.

Im folgenden wird das Register R6 als Korrektur-Register verwendet. Es wird auf einen Anfangswert gesetzt, der der halben Strecke von Delta-X entspricht (422-425). Durch den Aufruf der Funktion Nr. 11 ($OB) wird der Inhalt von Register R2 (d.h. Delta-X) nach R7 transferiert (426-427). Dieses Register wird als Zähler benutzt.

Als nächstes (429-433) wird bestimmt, ob der Punkt neben den vorigen Punkt gesetzt wird oder gleichzeitig auch noch in Y-Richtung verschoben werden muß, d.h. ob die Steigung für den zu setzenden Punkt gleich Null oder gleich 45 Grad ist.

Dazu wird (beim Zeichnen in X-Richtung) die Summe aus Delta-Y und dem Inhalt des Korrektur-Registers R6 gebildet. Das Ergebnis wird mit Delta-X verglichen. Ist die genannte Summe kleiner als Delta-X, so ist die Steigung gleich 0 oder 180 Grad. Ansonsten beträgt die Steigung 45 Grad, d.h. die Y-Koordinate ändert sich (435-438).

Ist die Summe aus Delta-Y und dem Korrektur-Register jedoch nicht kleiner als Delta-X, so ändert sich nur der X-Wert, und es wird nach DRX3 (439) verzweigt.

Mit den auf diese Weise geänderten Koordinaten wird dann der betreffende Punkt gesetzt bzw. gelöscht, je nachdem wie der Inhalt des COLOR-Bytes festgelegt wurde (441).

Anschließend wird die Weglänge dekrementiert (442-443) und wieder zum Label DRX2 (429) verzweigt (444). Diese Schleife wird solange durchlaufen, bis der Zähler für die Weglänge gleich Null ist.

Entsprechend ist das Programm DRAWY zum Zeichnen in Y-Richtung aufgebaut:

```
447:  C25E 20 00 C0 DRAWY   JSR  ASSI    ;KORREKTUR-REGISTER IST R6
448:  C261 0B 36            .BYTE$0B,$36  ;DY -> R6
449:  C263 20 00 C0         JSR  ASSI
450:  C266 05 60            .BYTE$05,$60  ;R6/2 -> KORREKTUR
451:  C268 20 00 C0         JSR  ASSI
452:  C26B 0B 37            .BYTE$0B,$37  ;R7 = ZAEHLER

454:  C26D 20 00 C0 DRY2    JSR  ASSI
455:  C270 0D 62            .BYTE$0D,$62  ;KORR+DX -> KORR
456:  C272 20 00 C0         JSR  ASSI
457:  C275 0C 63            .BYTE$0C,$63  ;VERGLEICHE KORR,DY
458:  C277 90 0A            BCC  DRY3    ;KLEINER, X-KOORD. BLEIBT
                      ; KORR>DY, BEIDE KOORD. ANDERN SICH
460:  C279 20 00 C0         JSR  ASSI
461:  C27C 0E 63            .BYTE$0E,$63  ;KORR-DY -> KORR
462:  C27E 20 00 C0         JSR  ASSI
463:  C281 0D 04            .BYTE$0D,$04  ;AENDERT X-KOORD.
464:  C283 20 00 C0 DRY3    JSR  ASSI
465:  C286 0D 15            .BYTE$0D,$15  ;AENDERT Y-KOORD.
466:  C288 20 93 C2         JSR  TOGGLE  ;SET OR CLEAR BIT
467:  C28B 20 00 C0         JSR  ASSI
468:  C28E 04 70            .BYTE$04,$70  ;WEG DEKREMENTIEREN
469:  C290 D0 DB            BNE  DRY2
470:  C292 60       DRFINI  RTS
```

7.2 16-Bit-Routinen zur Sprite-Programmierung

"Sprites" sind eine besondere Art von frei definierbaren Grafikzeichen, die in einem Raster von 24 Bildpunkten (horizontal) mal 21 Punkten (vertikal) erstellt werden können.

Diese Form der Grafikdarstellung wird durch einen speziellen Baustein des Commodore 64 unterstützt, den sogenannten Video-Chip. Eine fertiggestellte Sprite-Figur kann durch Angabe der Koordinaten leicht an eine beliebige Stelle des Bildschirms gebracht werden bzw. durch kontinuierliche Veränderung der Koordinaten über den Bildschirm bewegt werden.

Da zur Speicherung eines Bildschirmpunktes ein Bit benötigt wird, braucht man für ein Sprite 24 x 21 = 504 Bits, also 504 / 8 = 63 Bytes. Diese 63 Bytes kann man sich in 21 Zeilen zu je 3 Bytes wie folgt angeordnet vorstellen:

```
BYTE  0    BYTE  1    BYTE  2
BYTE  3    BYTE  4    BYTE  5
...        ...        ...
...        ...        ...
...        ...        ...

BYTE 60    BYTE 61    BYTE 62
```

Auf Bit-Ebene bedeutet es, daß ein Grafikpunkt eingeschaltet ist, wenn das entsprechende Bit gleich 1 ist, und ein Punkt ist ausgeschaltet, wenn das entsprechende Bit gleich 0 ist.

Zu den 63 Bytes für die Sprite-Definition kommt noch ein Byte hinzu, das als Platzhalter dient, so daß sich insgesamt 64 Bytes pro Sprite ergeben.

Insgesamt können bis zu 8 Sprites gleichzeitig auf dem Bildschirm dargestellt werden. Jedem dieser 8 Sprites ist ein Sprite-Zeiger (Pointer) zugeordnet, der aus einem Byte besteht. Diese 8 Zeiger befinden sich in in den letzten 8 Bytes des Bildschirmspeichers. Dies ist möglich, da von den 1024 Bytes des Bildschirmspeichers ja nur 1000 Bytes für den Bildschirm benötigt

werden (1000 = 40 Zeichen pro Zeile x 25 Zeilen). Die Sprite-Pointer liegen also normalerweise im Speicherbereich ab 2040 (dezimal) bzw. $07F8 (hexadezimal). Verschiebt man jedoch den Bildschirmspeicher, so verschiebt sich der Speicherbereich für die Sprite-Pointer ebenfalls entsprechend.

Steht im Speicherplatz 2042 (d.h. Sprite-Pointer Nr. 3) z.B. der Wert 13, so bedeutet dies, daß für die Sprite-Definition die 64 Bytes verwendet werden, die ab Adresse 13 * 64 = 832 stehen.

Das Ein- und Ausschalten der jeweils 8 möglichen Sprites wird durch den Speicherplatz $D015 (53269) kontrolliert, der dementsprechend auch als SPRITE ENABLE Register bezeichnet wird. Jedem Sprite ist hier ein Bit zugeordnet: ist dieses Bit gesetzt, so wird das entsprechende Sprite eingeschaltet, ist das Bit nicht gesetzt, wird das Sprite ausgeschaltet.

Um z.B. Sprite Nr. 0 einzuschalten, ohne daß dadurch der Zustand der anderen Sprites beeinflußt wird, muß der Inhalt von Speicher 53269 mit (dem einschließenden) ODER (OR) verknüpft und anschließend wieder in 53269 gespeichert werden. In BASIC kann dies allgemein so formuliert werden:

```
POKE 53269,PEEK(53269) OR (2↑SN)
```

wobei SN die Sprite-Nummer darstellt (0 bis 7).

Das Ausschalten eines Sprites kann über eine UND-Verknüpfung erfolgen:

```
POKE 53269,PEEK(53269) AND (255-2↑SN)
```

Die folgenden Routinen zeigen, wie man Sprites in Maschinensprache erstellen, ein- und ausschalten und positionieren kann.

Zu Beginn wird eine Anzahl von Speicherplätzen für bestimmte Aufgaben reserviert:

```
                    ;SPRITE-ROUTINEN

581:  C350          SPPMEM  =   $7F8    ;SPRITE POINTER MEMORY
582:  C350          SPPOSR  =   $D000   ;SPRITE POSITIONSREGISTER
583:  C350          SPCOLR  =   $D027   ;SPRITE COLOR REGISTER
584:  C350          SPENA   =   $D015   ;SPRITE ENABLE REGISTER
585:  C350          SPCHR   =   R0      ;SPRITE TABELLEN POINTER
586:  C350          SPMEM   =   R1      ;SPRITE MEMORY POINTER
587:  C350          LENGTH  =   R2      ;SPRITE ZEILENANZAHL
588:  C350          BITCNT  =   R3
589:  C350          SHIFTR  =   R3+1
                    ;
591:  C350 00       SPCOL   .BYTE$00
592:  C351 00       SPNUM   .BYTE$00
```

SPPMEM (SPRITE POINTER MEMORY: $7F8, dezimal 2040) ist die Anfangsadresse des acht Bytes großen Speicherbereiches, in dem die Sprite-Daten-Pointer für die acht gleichzeitig darstellbaren Sprites stehen.

SPPOSR (SPRITE POSITIONSREGISTER: $D000, dezimal 53248)) ist die Anfangsadresse der Sprite-Positions-Register, in denen die Sprite-Koordinaten in der Reihenfolge: X-, Y-Koordinate stehen. Da die X-Koordinate größer als 255 sein kann, wird zusätzlich ein Speicher für das höchstwertige Bit der X-Koordinaten benötigt (also für das 9. Bit).

SPCOLR (SPRITE COLOR REGISTER: $D027, dezimal 53287) ist die Anfangsadresse des Sprite-Farbspeichers. In diesem und den folgenden sieben Speichern steht die Farbe der Sprites von 0 bis 7.

SPENA (SPRITE ENABLE REGISTER: $D015, dezimal 53270) dient zum Ein- und Ausschalten eines Sprites.

Außerdem wurden noch die Register R0 bis R3 zusätzlich mit symbolischen Namen belegt, die ihre Funktion andeuten sollen.

In SPCOL ($C350) muß die Farbe und in SPNUM ($C351) die Nummer des gewünschten Sprites stehen.

7.2.1 Erstellen von Sprites

Das Erstellen von Sprites erfordert einen beträchtlichen Rechenaufwand, der von der im folgenden beschriebenen Routine MAKE SPRITE übernommen werden kann. Mit Hilfe dieses Programmes ist es möglich, die gewünschte Sprite-Figur einfach in grafischer Form einzugeben. Die notwendigen Berechnungen werden dann vom Programm durchgeführt. Um das Programm SPMAKE (MAKE SPRITE) zu benutzen, muß beim Aufruf von SPMAKE nur noch die Anfangsadresse der Tabelle mitangegeben werden, in der die Sprite-Definition steht.

Diese Tabelle muß natürlich vorher definiert worden sein. Zur Markierung eines Grafik-Punktes innerhalb des Sprite-Rasters kann ein beliebiges Zeichen verwendet werden; Punkte, die nicht gesetzt werden sollen, werden durch Leerzeichen gekennzeichnet. Eine Sprite-Definition kann damit z.B. wie folgt aussehen:

```
                    ;SPRITE-DEFINITIONSTABELLE

704:  C412 2A 2A 2A QUADER   .ASC "************************"
705:  C42A 2A 20 20          .ASC "*                        *"
706:  C442 2A 20 20          .ASC "*                        *"
707:  C45A 2A 20 20          .ASC "*                        *"
708:  C472 2A 20 20          .ASC "*    ####                *"
709:  C48A 2A 20 20          .ASC "*    ####                *"
710:  C4A2 2A 20 20          .ASC "*    ####                *"
711:  C4BA 2A 20 20          .ASC "*    ####                *"
712:  C4D2 2A 20 20          .ASC "*                        *"
713:  C4EA 2A 20 20          .ASC "*                        *"
714:  C502 2A 20 20          .ASC "*                        *"
715:  C51A 2A 20 20          .ASC "*                        *"
716:  C532 2A 20 20          .ASC "*                        *"
717:  C54A 2A 20 20          .ASC "*                        *"
718:  C562 2A 20 20          .ASC "*                        *"
719:  C57A 2A 20 20          .ASC "*                        *"
720:  C592 2A 20 20          .ASC "*                        *"
721:  C5AA 2A 20 20          .ASC "*                        *"
722:  C5C2 2A 20 20          .ASC "*                        *"
723:  C5DA 2A 20 20          .ASC "*                        *"
724:  C5F2 2A 2A 2A          .ASC "************************"
```

Die Sprite-Figur kann also direkt in grafischer Form innerhalb eines Sprite-Rasters von 21 Zeilen zu je 24 Spalten eingegeben werden, indem mit Hilfe des Assemblers ein String definiert wird, der in einer Tabelle gespeichert wird, die bei Adresse $COC6 beginnt.

Damit kommen wir nun zur Routine MAKE SPRITE (SPMAKE), die mit Hilfe der zuvor besprochenen Subroutinen das in der Sprite-Tabelle definierte Sprite

```
                        ;SPRITE ERSTELLEN (MAKE SPRITE)
640:  C39F 20 88 C0 SPMAKE  JSR  GETADR  ;ADRESSE DER ANFANGSADRESSE DER SPRITE-TABELLE HOLEN
641:  C3A2 20 97 C0         JSR  INCG
642:  C3A5 20 A7 C0         JSR  PUTADR  ;KORR. RUECKSPRUNGADRESSE ZURUECKSCHREIBEN
643:  C3A8 20 9E C0         JSR  DECG
644:  C3AB 20 B7 C0         JSR  LGIND   ;TABELLEN-ANFANGSADRESSE HOLEN UND ...
645:  C3AE A2 00            LDX  #$00    ;NACH R0 ...
646:  C3B0 20 48 C1         JSR  SAVREG  ;SPEICHERN
647:  C3B3 20 8B C3         JSR  SPPTR   ;SPRITE POINTER HOLEN
648:  C3B6 A0 00            LDY  #$00
649:  C3B8 A9 15            LDA  #$15    ;21 SPRITEZEILEN
650:  C3BA 85 4F            STA  LENGTH

652:  C3BC A2 18    SPM0    LDX  #$18    ;24 SPRITESPALTEN
653:  C3BE A9 08    SPM1    LDA  #$08
654:  C3C0 85 51            STA  BITCNT
655:  C3C2 84 52            STY  SHIFTR  ;SHIFT REGISTER LOESCHEN
656:  C3C4 B1 4B    SPM2    LDA  (SPCHR),Y  ;SPRITEZEICHEN AUS TABELLE HOLEN
657:  C3C6 E6 4B            INC  SPCHR   ;AUF NAECHSTES ZEICHEN ZEIGEN
658:  C3C8 D0 02            BNE  SPM22
659:  C3CA E6 4C            INC  SPCHR+1
660:  C3CC 29 7F    SPM22   AND  #$7F    ;MSB LOESCHEN
661:  C3CE C9 20            CMP  #$20    ;LEERZEICHEN STEHT FUER "SPRITE-PUNKT AUS"
662:  C3D0 F0 03            BEQ  SPM3
663:  C3D2 38               SEC
664:  C3D3 B0 01            BCS  SPM33
665:  C3D5 18       SPM3    CLC
666:  C3D6 26 52    SPM33   ROL  SHIFTR
667:  C3D8 CA       SPM4    DEX          ;NAECHSTER SPRITEPUNKT
668:  C3D9 F0 11            BEQ  SPM5    ;WENN JA, NAECHSTE SPRITEZEILE
669:  C3DB C6 51            DEC  BITCNT  ;NAECHSTES BIT
670:  C3DD D0 E5            BNE  SPM2    ;NAECHSTEN PUNKT IM SELBEN BYTE BEARBEITEN
671:  C3DF A5 52            LDA  SHIFTR  ;EIN BYTE FERTIG
672:  C3E1 91 4D            STA  (SPMEM),Y
673:  C3E3 E6 4D            INC  SPMEM
674:  C3E5 D0 02            BNE  SPM44
675:  C3E7 E6 4E            INC  SPMEM+1
676:  C3E9 4C BE C3 SPM44   JMP  SPM1
677:  C3EC C6 51    SPM5    DEC  BITCNT
678:  C3EE A5 52            LDA  SHIFTR
679:  C3F0 91 4D            STA  (SPMEM),Y
680:  C3F2 E6 4D            INC  SPMEM
681:  C3F4 D0 02            BNE  SPM55
682:  C3F6 E6 4E            INC  SPMEM+1
683:  C3F8 C6 4F    SPM55   DEC  LENGTH
684:  C3FA D0 C0            BNE  SPM0
685:  C3FC 60               RTS
```

Zuerst wird die Adresse der Anfangsadresse der Sprite-Tabelle geholt, die im Benutzer-Programm nach dem Aufruf von SPMAKE stehen muß (640). Da der Rücksprung anschließend nicht wieder zu dieser Adresse geleitet werden soll, muß durch Inkrementieren eine entsprechende Korrektur erfolgen, damit nach dem Rücksprung der nächste Befehl ausgeführt wird (641).

Da zuvor die Rücksprungadresse inkrementiert wurde, muß sie nach dem Aufruf von PUTADR (642) wieder dekrementiert werden (643).

Anschließend steht die Adresse der Anfangsadresse der Tabelle im General-Register. Um die eigentliche Adresse der Tabelle zu erhalten, benutzen wir das Unterprogramm LGIND zum indirekten Laden des General-Registers (644).

Danach wird die Tabellenadresse in RO gespeichert. Dazu wird das X-Register mit $00 geladen, d.h. dem Pointer auf RO (645) und anschließend die Routine SAVREG aufgerufen (646).

Mit dem Aufruf von SPPTR (GET SPRITE POINTER) wird der Zeiger auf das Sprite geholt (647).

Als nächstes müssen die ASCII-Zeichen für die Sprite-Definition (704-724) in die benötigten Sprite-Daten umgerechnet werden. Dazu wird zunächst das Y-Register für die indirekte Adressierung mit $00 geladen (648).

Da die Tabelle 21 ($15) Zeilen enthält, wird dieser Wert in den Akku gebracht (649) und von da aus ins LENGTH-Register (650).

Die Zahl der Spalten (24 bzw. $18) wird ins X-Register geladen (652).

Die Berechnung der Sprite-Punkte erfolgt nun zeilenweise. Da es sich um 21 Zeilen handelt, wird dazu eine Schleife 21 mal durchlaufen (652-684). In Zeile 683 wird der Wert des Längenbytes LENGTH - das zuvor mit 21 belegt wurde - bei jedem Durchgang um eins vermindert (dekrementiert).

Wie schon weiter oben dargestellt wurde, werden für die Speicherung einer Spritezeile drei Bytes be-

nötigt (3 = 24/8), d.h. aus jeweils 8 ASCII-Zeichen muß jetzt ein Sprite-Datenbyte errechnet werden. Deshalb wird nun die Zahl 8 in den Speicher BITCNT (BIT COUNT = Low-Byte von R3, vgl. 588) geladen (653-654). Das Shift-Register SHIFTR (= High-Byte von R3, vgl. 589) wird gelöscht, indem der Wert $OO aus dem Y-Register nach SHIFTR gespeichert wird (655).

Bei der Markierung SPM2 werden dann die einzelnen Zeichen aus der Zeile geholt und nach SPCHR (=RO, vgl. 585) gebracht (656). Anschließend wird der Tabellenzeiger inkrementiert, der nun auf das nächste Zeichen zeigt (657). Da dieser Zeiger aus zwei Bytes besteht, müssen wir noch das High-Byte berücksichtigen (658-659).

Da wir wissen, daß das Zeichen, das soeben (in Zeile 656) gelesen wurde, im Akku steht, kann die folgende Operation direkt auf den Akku angewendet werden: aufgrund des spezifischen Commodore-Zeichensatzes wird mit dem jeweiligen Zeichencode eine UND-Verknüpfung mit $7F (127 dezimal) durchgeführt, d.h. das höchstwertige Bit wird gelöscht (660). Damit wird erreicht, daß der Zeichencode innerhalb des ASCII-Bereiches (von O-127) bleibt.

Dann wird überprüft, ob es sich bei dem Zeichen um ein Leerzeichen (ASCII-Code $2O, 32 dezimal) handelt (das ja bedeuten soll, daß der betreffende Sprite-Punkt nicht eingeschaltet sein soll) (661). Ist dies der Fall, so wird nach SPM3 (665) verzweigt (662), wo das Carry-Bit gelöscht wird. Handelte es sich jedoch um kein Leerzeichen, soll also der betreffende Sprite-Punkt gesetzt werden, so wird das Carry-Bit gesetzt (663) und nach SPM33 verzweigt (664).

Mit dem ROL-Befehl wird das Carry-Bit anschließend ins Shift-Register SHIFTR gebracht (666).

Jetzt wird der nächste Sprite-Punkt (Pixel) bearbeitet. Nachdem X (also der Spaltenzähler) dekrementiert wurde, kann man feststellen, ob die Zeile zu Ende ist. Dies ist der Fall, wenn X gleich Null ist. In dem Fall wird nach SPM5 (677(zur Bearbeitung der nächsten Zeile verzweigt (668).

Ist die aktuelle Zeile noch nicht abgearbeitet, so wird der Bitzähler BITCNT dekrementiert, um an das nächste Bit zu gelangen (669). Ist der Bitzähler ungleich Null, so ist das aktuelle Byte noch nicht abgearbeitet, und es wird zurück verzweigt nach SPM2 (656), wo das nächste ASCII-Zeichen aus der Sprite-Definitions-Tabelle geholt wird (670).

Nach jedem vollständigen Byte wird der Inhalt des Shift-Registers SHIFTR mit Y indiziert in den Spritespeicher SPMEM gebracht (671-672). Danach muß der Pointer innerhalb des Spritespeichers inkrementiert werden (673). Da es sich um ein 16-Bit-Register handelt, muß auch das High-Byte berücksichtigt werden (674-675). Nach dem Inkrementieren erfolgt eine Verzweigung zurück nach SPM1 (653), um die nächsten 8 Zeichen zu bearbeiten (676).

Zu der Markierung SPM5 (677) wird dann verzweigt (668), wenn eine Zeile vollständig abgearbeitet ist: der Bitzähler BITCNT wird dekrementiert und der Inhalt von SHIFTR nach SPMEM gebracht (677-682). Anschließend wird der Zeilenzähler LENGTH dekrementiert (683). Falls er noch ungleich Null ist, also noch nicht alle Zeilen abgearbeitet sind, wird wieder zurück verzweigt (684) nach SPM0 (652). Ansonsten ist die Sprite-Figur fertig und die Routine MAKE SPRITE beendet (685).

7.2.2 Sprite-Positionierung

```
                        ;SPRITE POSITIONIERUNG
688:  C3FD              XKOR    =    R4
689:  C3FD              YKOR    =    R5
690:  C3FD AD 51 C3 SPPOS   LDA  SPNUM
691:  C400 0A               ASL
692:  C401 A8               TAY
693:  C402 A5 55            LDA  YKOR
694:  C404 99 01 D0         STA  SPPOSR+1,Y
695:  C407 A5 53            LDA  XKOR
696:  C409 99 00 D0         STA  SPPOSR,Y
697:  C40C A5 54            LDA  XKOR+1
698:  C40E 8D 10 D0         STA  SPPOSR+16
699:  C411 60               RTS
```

Das Sprite wird an die Bildschirmposition mit den Koordinaten XKOR und YKOR gebracht. Für die X-Koordinate wird das Register R4, für die Y-Koordinate das Register R5 verwendet.

Zunächst wird die Spritenummer geholt (690). Anhand der Spritenummer werden dann die Koordinaten ins Sprite-Positions-Register geladen. (693-698). Da die Koordinaten aus jeweils zwei Bytes bestehen, wird die Spritenummer verdoppelt (691), bevor sie im Y-Register als Index benutzt wird (692). - Die folgende Tabelle gibt einen Überblick über die Sprite-Positionierungs-Register:

Adresse		Inhalt
$D000	53248	Sprite 0 X-Koordinate
$D001	53249	Sprite 0 Y-Koordinate
$D002	53250	Sprite 1 X-Koordinate
$D003	53251	Sprite 1 Y-Koordinate
$D004	53252	Sprite 2 X-Koordinate
$D005	53253	Sprite 2 Y-Koordinate
...		
...		
$D00E	53262	Sprite 7 X-Koordinate
$D00F	53263	Sprite 7 Y-Koordinate
$D010	53264	Bit 9 für X-Koord. Sprites 0-7

7.3 Weitere Routinen zur Sprite-Programmierung

7.3.1 Ein- und Ausschalten von Sprites

```
                    ;SPRITE ON, SPRITE# IN (SPNUM)
595:  C352 AD 51 C3 SPRON   LDA  SPNUM   ;SPRITENUMMER HOLEN
596:  C355 20 77 C3         JSR  SETPOS  ;BIT-POSITION ENTSPR. SPRITENR. IN DEN AKKU
597:  C358 0D 15 D0         ORA  SPENA   ;SPRITE ...
598:  C35B 8D 15 D0         STA  SPENA   ;EINSCHALTEN

                    ;SPRITEFARBE BESTIMMEN (SET SPRITE COLOR)
601:  C35E AC 51 C3 SSPCOL  LDY  SPNUM   ;SPRITEFARBE IN DEN ...
602:  C361 AD 50 C3         LDA  SPCOL   ;DEM SPRITE ZUGEORDNETEN ...
603:  C364 99 27 D0         STA  SPCOLR,Y ;FARBSPEICHER BRINGEN
604:  C367 60               RTS
```

```
                    ;SPRITE AUSSCHALTEN (SPRITE OFF), SPRITE-NR. IN (SPNUM)
607:  C368 AD 51 C3 SPOFF   LDA  SPNUM   ;SPRITENUMMER HOLEN
608:  C36B 20 77 C3         JSR  SETPOS  ;BIT-POSITION ENTSPR. SPRITENR. IN DEN AKKU
609:  C36E 49 FF            EOR  #$FF    ;SPRITE ...
610:  C370 2D 15 D0         AND  SPENA   ;AUS-
611:  C373 8D 15 D0         STA  SPENA   ;SCHALTEN
612:  C376 60               RTS
```

Die Spritenummer, die in SPNUM stehen muß, wird in den Akku geladen (595). Darauf wird die Routine SETPOS (616-623) aufgerufen, in der anhand der Spritenummer die entsprechende Bitposition bestimmt wird, indem die Berechnung 2↑Spritenummer durchgeführt wird. Dazu wird der mit 1 initialisierte Inhalt des Akkus so oft verdoppelt, wie es der Spritenummer entspricht:

```
                    ;SET BIT IN ACCU
                    ;RETURN MIT BITPOSITION (=2↑SPRITENUMMER) IM AKKU
616:  C377 A8       SETPOS  TAY
617:  C378 A9 01            LDA  #$01
618:  C37A C8               INY
619:  C37B 88       SETP1   DEY          ;SPRITE-NR. HERUNTERZAEHLEN
620:  C37C F0 03            BEQ  SETPX   ;SOLANGE SPRITE-NR. <> 0 ...
621:  C37E 0A               ASL          ;AKKUINHALT (BITPOSITION) VERDOPPELN
622:  C37F D0 FA            BNE  SETP1
623:  C381 60       SETPX   RTS
```

Bei Spritenummer 3 würde also der Wert 2↑3 = 8 berechnet und über den Akku der aufrufenden Routine (SPRITE ON bzw. SPRITE OFF) übergeben.

Zum Einschalten eines Sprites wird der Inhalt des Akkus dann mit dem einschließenden ODER (OR) mit dem SPRITE ENABLE Register SPENA ($D015, also dezimal 53269) verknüpft (597), und das Ergebnis wird ins SPRITE ENABLE Register gebracht (598). Anschließend wird die Routine zum Setzen der Sprite-Farbe durchlaufen (601-604, s.u.).

Zum Ausschalten eines Sprites muß das diesem Sprite zugeordnete Bit gelöscht werden. Dies geschieht dadurch, daß (nachdem die Bitposition in SETPOS bestimmt worden ist) zunächst mit Hilfe der ausschließenden ODER-Verknüpfung (EOR) mit $FF der Wert dieses Bytes invertiert wird (609) und anschließend mit dem

Inhalt des SPRITE ENABLE Registers UND-verknüpft wird (610). Das Ergebnis wird dann ins SPRITE ENABLE Register gebracht (611). - Hierzu ein Beispiel:

Angenommen, die Sprites Nr. 1, 4, 6 und 7 sind eingeschaltet. Dann sieht der Inhalt des SPRITE ENABLE Registers so aus:

```
SPENA     1 1 0 1 0 0 1 0
```

Jetzt soll Sprite Nr. 1 (entsprechend Bit 1) ausgeschaltet werden. Nachdem in SETPOS die Bitposition bestimmt ist, steht also im Akku:

```
AKKU      0 0 0 0 0 0 1 0
```

Verknüpft man diesen Wert mit der exlusiven ODER-Funktion (EOR) mit $FF, so erhält man:

```
AKKU      0 0 0 0 0 0 1 0
$FF       1 1 1 1 1 1 1 1
--------------------------
AKKU      1 1 1 1 1 1 0 1
```

Durch die UND-Verknüpfung mit dem Inhalt des SPRITE ENABLE Registers wird schließlich das gewünschte Bit 1 gelöscht, ohne daß die übrigen Bits verändert werden:

```
AKKU      1 1 1 1 1 1 0 1
SPENA     1 1 0 1 0 0 1 0
--------------------------
AKKU      1 1 0 1 0 0 0 0
```

7.3.2 Bestimmung der Sprite-Farbe

```
                    ;SPRITEFARBE BESTIMMEN (SET SPRITE COLOR)
601:  C35E AC 51 C3 SSPCOL  LDY  SPNUM   ;SPRITEFARBE IN DEN ...
602:  C361 AD 50 C3         LDA  SPCOL   ;DEM SPRITE ZUGEORDNETEN ...
603:  C364 99 27 D0         STA  SPCOLR,Y ;FARBSPEICHER BRINGEN
604:  C367 60               RTS
```

Durch indizierte Adressierung mit Y wird die in SPCOL befindliche Farbnummer in den für das betreffende Sprite vorgesehenen Speicherplatz gebracht. Dieser Speicherbereich beginnt bei SPCOLR, also $D027 (vgl. Zeile 583).

Die folgende Tabelle enthält die Adressen der Farbspeicher für alle 8 Sprites:

Adresse		Inhalt
53287	$D027	Farbregister für Sprite 0
53288	$D028	Farbregister für Sprite 1
53289	$D029	Farbregister für Sprite 2
53290	$D02A	Farbregister für Sprite 3
53291	$D02B	Farbregister für Sprite 4
53292	$D02C	Farbregister für Sprite 5
53293	$D02D	Farbregister für Sprite 6
53294	$D02E	Farbregister für Sprite 7

7.3.3 Bestimmung der Anfangsadresse von Sprites

```
                    ;GET SPRITE POINTER
630:  C38B AC 51 C3 SPPTR   LDY  SPNUM
631:  C38E B9 88 C3         LDA  SPBLCK,Y  ;POINTER AUF SPRITE-DEFINITIONSBLOCK ...
632:  C391 99 F8 07         STA  SPPMEM,Y  ;INS SPRITE POINTER MEMORY
633:  C394 B9 85 C3         LDA  SPLTAB,Y  ;ANFANGSADRESSE DER SPRITE-DEFINITIONSTABELLE
634:  C397 85 4D            STA  SPMEM   ;INS SPRITE MEMORY POINTER REGISTER
635:  C399 B9 82 C3         LDA  SPHTAB,Y
636:  C39C 85 4E            STA  SPMEM+1
637:  C39E 60               RTS
```

Mit dieser Routine wird die Anfangsadresse eines Sprites bestimmt.

Dieser Sprite-Pointer kann mit Hilfe der Spritenummer ermittelt werden, da die Speicherung eines Sprites in Abhängigkeit von der Spritenummer erfolgt.

Wenn man nur 1 bis 3 Sprites verwenden will, kann man zur Speicherung der Sprite-Definitionen den Kassettenpuffer verwenden, der von 832 bis 1023 (dezimal) reicht.

Da wir in der gleich zu besprechenden Benutzer-Routine (Kap. 7.4.1) nur ein Sprite verwenden, können wir zur Speicherung der Sprite-Definitions-Blöcke den Speicherbereich des Kassettenpuffers benutzen. Für die in diesem Speicherbereich möglichen drei Sprites ergibt sich für Sprite Nr. 0 der Pointer 13, da 13 x 64 =

832 ist, also die Anfangsadresse des Kassettenpuffers. Entsprechend lauten die Pointer für Sprite 1 und 2: 14 bzw. 15. Hexadezimal ergeben sich also die drei Pointerwerte $0D, $0E, $0F (627).

Wie schon erwähnt, müssen diese Pointer am Ende des Bildschirmspeicherbereiches stehen, normalerweise also ab Adresse 2040. In diesen Speicher muß also in unserem Beispiel der Pointer 13 für Sprite Nr. 1 gebracht werden.

Die folgende Tabelle enthält außerdem die Adressen der drei Sprites, aufgeteilt in High- und Low-Byte: zu Sprite 0 gehört also die Adresse $0340 (dezimal 832), zu Sprite 1 die Adresse $0380 (dezimal 896) und zu Sprite 2 die Adresse $03C0 (dezimal 960):

```
625:  C382 03 03 03 SPHTAB   .BYTE$03,$03,$03
626:  C385 40 80 C0 SPLTAB   .BYTE$40,$80,$C0
627:  C388 0D 0E 0F SPBLCK   .BYTE$0D,$0E,$0F
```

Kommen wir zurück zur Routine GET SPRITE POINTER (SPPTR)! Anhand der Spritenummer wird der Akku mit dem Pointer auf den Definitions-Block für dieses Sprite geladen (631), der dann - ebenfalls anhand der Spritenummer - nach SPPMEM (SPRITE POINTER MEMORY) gebracht wird (632). Anschließend werden Low- und High-Byte der Anfangsadresse der Sprite-Definitions-Tabelle nach SPMEM bzw. SPMEM+1 (SPRITE MEMORY POINTER) gebracht (633-636).

7.4 Anwendungsbeispiele zur Benutzung der Sprite- und Grafik-Routinen

Im folgenden wird an zwei einfachen Beispielen gezeigt, wie sich die Sprite- und Grafik-Routinen in eigenen Programmen verwenden lassen. Die Steuerung der Programme erfolgt durch eine kleine Routine, die auf einen Tastendruck wartet:

```
                    ;WARTEN AUF TASTENDRUCK
727:  C60A 20 E4 FF KBDWT    JSR  GETIN
728:  C60D C9 00             CMP  #$00
729:  C60F F0 F9             BEQ  KBDWT
730:  C611 60                RTS
```

7.4.1 Beispiel zur Benutzung der Sprite-Routinen

Die folgende Routine illustriert die Verwendung der vorgestellten Sprite-Routinen.

```
                    ;SPRITE-BENUTZERPROGRAMM
732:  C612 20 E7 C0         JSR  CLAREG  ;ALLE REGISTER LOESCHEN
733:  C615 20 04 C3         JSR  GON     ;GRAFIK EINSCHALTEN
734:  C618 A9 02            LDA  #$02    ;HINTERGRUNDFARBE ...
735:  C61A 20 3D C3         JSR  GCOL    ;BESTIMMEN
736:  C61D 20 2A C3         JSR  GCLR    ;GRAFIKSPEICHER LOESCHEN
737:  C620 A9 02            LDA  #$02    ;SPRITENUMMER
738:  C622 8D 51 C3         STA  SPNUM
739:  C625 A9 07            LDA  #$07    ;SPRITEFARBE
740:  C627 8D 50 C3         STA  SPCOL
741:  C62A 20 9F C3         JSR  SPMAKE  ;SPRITE ERSTELLEN
742:  C62D 12 C4            .WORDQUADER  ;DEFINITIONSTABELLE BEGINNT BEI QUADER
743:  C62F A9 64            LDA  #$64    ;SPRITEKOORDINATEN
744:  C631 85 53            STA  R4      ;X-KOORDINATE
745:  C633 85 55            STA  R5      ;Y-KOORDINATE
746:  C635 20 52 C3         JSR  SPRON   ;SPRITE EINSCHALTEN
747:  C638 20 FD C3         JSR  SPPOS   ;SPRITE POSITIONIEREN
748:  C63B 20 0A C6         JSR  KBDWT   ;WARTEN AUF TASTENDRUCK
749:  C63E 20 68 C3         JSR  SPOFF   ;SPRITE AUSSCHALTEN
750:  C641 20 1B C3         JSR  GOFF    ;GRAFIK AUSSCHALTEN
```

Nachdem sicherheitshalber alle Register gelöscht worden sind (732), wird mit dem Aufruf des Unterprogrammes GON der Grafik-Modus eingeschaltet (733). Für die Hintergrundfarbe wird der Wert $02 gewählt (734). Das Unterprogramm GCOL setzt die Hintergrundfarbe (735), und mit dem Unterprogramm GCLR wird der Grafik-Bildschirm gelöscht (736).

Es soll das Sprite Nr. 2 definiert werden. Deshalb wird diese Spritenummer nach SPNUM geladen (737-738).

Die Spritefarbe soll dem Farbwert $07 entsprechen, der nach SPCOL gebracht wird (739-740).

Mit dem Aufruf der Routine SPMAKE sowie der Angabe der Adresse der Sprite-Definitions-Tabelle wird das Sprite erstellt (741-742). Nachdem die Sprite-Koordinaten in die dafür vorgesehenen Speicher gebracht worden sind (743-745), kann das Sprite schließlich an der gewünschten Position eingeschaltet werden (746-747).

Die Darstellung bleibt nun solange auf dem Bildschirm, bis eine beliebige Taste gedrückt wird

(748). In der Routine KBDWT (KeyBoard WaiT) wird übrigens eine interne ROM-Routine des Commodore 64 benutzt. Auf die Verwendung solcher Routinen kommen wir noch zu sprechen.

7.4.2 Beispiel zur Benutzung der Grafik-Routinen

Das folgende Programm zeigt eine Anwendung der Grafik-Routinen: mit Hilfe der vorgestellten Routine PLOT wird eine Linie auf dem Bildschirm gezeichnet. In unserem Demonstrations-Programm folgt es direkt auf die Sprite-Darstellung: nachdem eine Taste gedrückt wird (748), werden das Sprite und die Grafik-Darstellung ausgeschaltet (749-750).

```
                    ;GRAFIK-BENUTZERPROGRAMM
753:  C641 20 04 C3         JSR  GON     ;GRAFIK EINSCHALTEN
754:  C644 A9 05            LDA  #$05    ;HINTERGRUNDFARBE ...
755:  C646 20 3D C3         JSR  GCOL    ;BESTIMMEN
756:  C649 20 2A C3         JSR  GCLR    ;GRAFIKSPEICHER LOESCHEN

                    ;LINIE (0,0 - 319,199)
759:  C64C 20 00 C0         JSR  ASSI
760:  C64F 20 00 00         .BYTE$20,$00,$00,$00  ;XS=0
761:  C653 20 00 C0         JSR  ASSI
762:  C656 0B 01            .BYTE$0B,$01  ;YS=XS
763:  C658 20 00 C0         JSR  ASSI
764:  C65B 20 02 3F         .BYTE$20,$02,$3F,$01  ;XE=319
765:  C65F 20 00 C0         JSR  ASSI
766:  C662 20 03 C7         .BYTE$20,$03,$C7,$00  ;YE=199
767:  C666 20 CE C1         JSR  PLOT    ;LINIE ZEICHNEN
768:  C669 20 0A C6         JSR  KBDWT   ;WARTEN AUF TASTENDRUCK
769:  C66C 20 1B C3         JSR  GOFF    ;GRAFIK AUSSCHALTEN
770:  C66F 60               RTS
```

Die Grafik-Darstellung wird eingeschaltet (753), die Hintergrundfarbe bestimmt (754-755) und der Grafikspeicher gelöscht (756).

Mit Hilfe der Funktion Nr. 32 werden die Anfangs- und End-Koordinaten der Geraden in die dafür vorgesehenen Speicher gebracht (759-766). Nachdem die Gerade gezeichnet worden ist (767), wartet das Programm wieder auf einen Tastendruck (768).

8 | Verwendung von ROM-Routinen

8. Verwendung von ROM-Routinen

8.1 Parameterübergabe von BASIC-Programmen aus

Mit dieser Routine kann man von einem BASIC-Programm aus mit dem BASIC-Befehl SYS Parameter an ein Maschinenprogramm übergeben. Gezeigt wird das Verfahren am Beispiel zweier Parameter, grundsätzlich können auf diese Weise jedoch auch noch mehr Parameter übergeben werden.

Das Programm wird folgendermaßen benutzt: Nach der BASIC-Anweisung SYS wird - wie üblich - die Anfangsadresse des Maschinenpogramms angegeben. Danach folgt, durch ein Komma getrennt, der erste zu übergebende Parameter. Dies muß übrigens kein einfacher Wert sein, sondern es kann auch ein Ausdruck angegeben werden, der noch berechnet werden muß.

Beispiele zur Parameterübergabe:

```
SYS 51200,880
SYS 51200,16000+400
```

Für die Routine zur Parameterübergabe werden die folgenden Vereinbarungen getroffen:

```
                    ; FILENAME..... ROM-ROUTINEN
                    ; LAST UPDATE.. 23-AUG-84
                    ; TRENG
 4:     C800        CHRIN    =    $FFCF
 5:     C800        CHROUT   =    $FFD2
 6:     C800        GETIN    =    $FFE4
 7:     C800        GETWRD   =    $B7F7
 8:     C800        GETBYT   =    $B79E
 9:     C800        CHKOM    =    $AEFD
 10:    C800        FORMEL   =    $AD8A

 12:    C800        CR       =    $0D
 13:    C800        LF       =    $0A
 14:    C800        BLANK    =    $20
 15:    C800        PROMPT   =    ")"
```

In den Zeilen 4-10 sind die Anfangsadressen interner ROM-Routinen des Commodore 64 angegeben, die beim Programmieren in Maschinensprache genau so wie eigene Unterprogramme verwendet werden können.

CHRIN holt ein Zeichen aus dem Eingabekanal, im algemeinen von der Tastatur, falls keine andere Zuordordnung getroffen wurde. Die Anfangsadresse dieser Routine ist $FFCF (65487 dezimal).

CHROUT dient zur Ausgabe eines Zeichens auf den Bildschirm, sofern kein anderer Ausgabekanal vereinbart wurde. Das auszugebende Zeichen muß im Akku stehen, bevor diese Routine aufgrufen wird. Die Anfangsadresse ist $FFD2 (65490 dezimal).

GETIN bringt den ASCII-Wert eines von der Tastatur gelesenen Zeichens in den Akku. Die Anfangsadresse ist $FFE4 (65508 dezimal).

GETWRD liest ein Datenwort, also zwei Bytes ein. Dieser Wert steht anschließend in den Adressen $14 und $15. Die Anfangsadresse ist $B7F7 (47095 dezimal).

GETBYT liest ein Byte ein, das danach im X-Register zur Verfügung steht. Die Anfangsadresse ist $B79E (47006 dezimal).

CHKOM überprüft, ob das nächste Zeichen ein Komma ist. Die Anfangsadresse ist $AEFD (44797 dezimal).

FORMEL wertet einen zu berechnenden Ausdruck aus. Die Anfangsadresse ist $AD8A (44426 dezimal).

Weiterhin werden noch einige spezielle ASCII-Zeichen vereinbart: CR für Carriage Return, LF für Line Feed, BLANK für das Leerzeichen und " " als sogenanntes Prompt-Zeichen, das anzeigen soll, daß vom Programm eine Eingabe erwartet wird.

Ein kurzes Programm soll die Verwendung der vorgestellten ROM-Routinen illustrieren. Mit diesem Programm können von BASIC aus zwei Parameter an ein Maschinenspracheprogramm übergeben werden, von denen der erste ein positiver, ganzzahliger Ausdruck im Bereich von 0 bis 2↑16-1 sein darf. Der zweite Parameter kann eine positive, ganze Zahl oder eine Variable im Bereich von 0 bis 255 sein.

```
                        ;PARAMETER-UEBERGABE MIT SYS
18:    C800 20 FD AE SYSPAR   JSR  CHKOM
19:    C803 20 8A AD          JSR  FORMEL
20:    C806 20 F7 B7          JSR  GETWRD
21:    C809 A5 15             LDA  $15
22:    C80B A6 14             LDX  $14
23:    C80D 20 80 C8          JSR  PRADR
24:    C810 20 FD AE          JSR  CHKOM
25:    C813 20 9E B7          JSR  GETBYT
26:    C816 8A                TXA
27:    C817 4C 6A C8          JMP  PRBYTE
```

Die ROM-Routine CHKOM überprüft, ob auf die nach SYS angegebene Adresse ein Komma folgt, d.h. ob überhaupt ein Parameter übergeben werden soll (18).

Damit auch noch zu berechnende Ausdrücke als Paramter übergeben werden können, wird die ROM-Routine FORMEL aufgerufen, die den hinter dem Komma angegebenen Ausdruck gegebenenfalls auswertet (19).

Die ROM-Routine GETWRD wandelt den von der FORMEL-Routine ausgewerteten Parameter in die Binärdarstellung um (20). Das Ergebnis steht in den Adressen $14 und $15. Nachdem der Inhalt dieser Speicher in den Akku bzw. ins X-Register geladen wurde (21-22), wird diese Adresse durch das Unterprogramm PRADR in Hexadezimaldarstellung ausgegeben (23).

Danach wird überprüft, ob noch ein Komma folgt (24). Mit der ROM-Routine GETBYT wird dann ein Byte eingelesen, das anschließend im X-Register zur Verfügung steht (25) und von da aus in den Akku gebracht werden kann (26). Mit der Routine PRBYTE wird dieses Byte in Hexadezimaldarstellung ausgegeben (27). Der zweite Parameter in unserem Programm-Beispiel darf also nur ein einziges Zeichen sein.

Möchte man mehrere Parameter übergeben, so müßte der entsprechende Programmteil mit einer Schleife mehrmals durchlaufen werden, entsprechend der Anzahl der Parameter.

Im folgenden werden die von dieser Routine verwendeten Unterprogramme vorgestellt. Zunächst das Programm zur Ausgabe des Akku-Inhaltes in hexadezimaler Darstellung:

```
                        ;BYTE HEXADEZIMAL AUSGEBEN
84:   C86A 48       PRBYTE   PHA
85:   C86B 4A                LSR  A
86:   C86C 4A                LSR  A
87:   C86D 4A                LSR  A
88:   C86E 4A                LSR  A
89:   C86F 20 75 C8          JSR  PRHEX1
90:   C872 68                PLA
91:   C873 29 0F    PRHEX    AND  #$0F
92:   C875 09 30    PRHEX1   ORA  #$30
93:   C877 C9 3A              CMP  #$3A     ;":"
94:   C879 90 02              BCC  PRHEX2
95:   C87B 69 06              ADC  #$06
96:   C87D 4C D2 FF PRHEX2    JMP  CHROUT
```

Sofern der Wert eines Bytes einem ASCII-Zeichen für eine Ziffer entspricht (ASCII-Code $30-$39 bzw. 48-57 dezimal), wird dieses Zeichen sofort ausgegeben. Ansonsten erfolgt die Ausgabe eines Buchstaben-Zeichens von A bis F.

Der Akku-Inhalt wird zunächst auf dem Stack gesichert (84).

Da der Inhalt eines Bytes durch zwei Hexadezimalzeichen dargestellt werden kann (von $00 - $FF), wird jetzt zuerst das höherwertige Nibble (also die obere Byte-Hälfte) und dann das niederwertige Nibble (die untere Byte-Hälfte) in ein Hexadezimalzeichen umgewandelt. Durch viermaliges Verschieben des Akku-Inhaltes um eine Bit-Position nach rechts wird das höherwertige Nibble in die untere Byte-Hälfte verschoben (85-88). Damit erhalten wir einen Binärwert von 0 bis 15.

Durch Addition von $30 (48 dezimal) ergibt sich für die Ziffern von 0 bis 9 der zugehörige ASCII-Code. Da wir jedoch wissen, daß der Binärwert zwischen 0 und 15 liegt, können wir dieses Ergebnis auch durch eine ODER-Verknüpfung mit $30 erhalten (92).

Da im ASCII-Code auf die Ziffern-Zeichen nicht sofort die Buchstaben-Zeichen folgen, sondern andere Sonderzeichen, müssen diese Zeichencodes übersprungen werden, um an die Buchstaben-Codes zu gelangen. Soll also ein Hexadezimalzeichen von A bis F ausgegeben werden, muß der Wert 7 zum ASCII-Code hinzuaddiert werden, um zum ersten Buchstaben-Code zu gelangen. Dies

geschieht wie folgt: wird bei dem Vergleich des ermittelten ASCII-Zeichen-Codes festgestellt, daß der Code kleiner ist als $3A (58 dezimal), so handelt es sich um eine Ziffer, die sofort ausgegeben werden kann (93-94). Ist der ASCII-Code jedoch größer oder gleich 58, so muß 7 hinzuaddiert werden, um den richtigen Buchstaben-Code zu erreichen. Da der Additionsbefehl ADC eine Addition mit Carry bedeutet und das Carry-Bit in dem Fall gesetzt ist, darf jedoch nur noch 6 addiert werden (95). Anschließend wird mit der ROM-Routine CHROUT das betreffende Buchstaben-Zeichen ausgegeben (96).

```
                        ;PRINT ADRESSE
                        ;HIGH BYTE IN A
                        ;LOW BYTE IN X
101:   C880 20 6A C8 PRADR    JSR  PRBYTE
102:   C883 8A               TXA
103:   C884 4C 6A C8         JMP  PRBYTE
```

Dieses Unterprogramm übergibt die beiden Bytes im Akku und im X-Register der Reihe nach dem Unterprogramm PRBYTE, das den Inhalt des Akkus in hexadezimaler Darstellung ausgibt. Das High-Byte muß im Akku, das Low-Byte im X-Register stehen.

8.2 Weitere Beispiele mit ROM-Routinen

```
                        ;EINGABE
30:    C81A A9 3E    INPUT    LDA  #PROMPT
31:    C81C 20 D2 FF          JSR  CHROUT
32:    C81F A0 00             LDY  #$00
33:    C821 20 CF FF READ     JSR  CHRIN
34:    C824 99 87 C8          STA  BUFF,Y
35:    C827 C8                INY
36:    C828 C9 0D             CMP  #CR
37:    C82A D0 F5             BNE  READ
38:    C82C 4C 3D C8          JMP  CROUT
```

Mit dieser Routine kann ein Text eingegeben werden, der in dem dafür vorgesehenen Speicherbereich ab Adresse $C887 gespeichert wird:

```
105:   C887 00        BUFF      .BYTE$00
```

Beim Aufruf des Programms wird zunächst das vorher (15) definierte Promptzeichen ">" ausgegeben. Dazu wird dieser Zeichenwert in den Akku gebracht (30) und von da mit Hilfe der ROM-Routine CHROUT ausgegeben (31).

Da die nun folgenden Eingabezeichen ab Adresse $C887 gespeichert werden sollen, wird das Index-Register Y gelöscht, damit es auf den Anfang dieses Speicherbereichs zeigt (32).

In der folgenden Schleife wird jeweils ein Zeichen eingegeben und in dem reservierten Speicherbereich abgelegt (33-37). Zur Eingabe jedes Zeichens wird die ROM-Routine CHRIN aufgerufen (33). Dann wird das Zeichen in den Speicher gebracht (34) und der Index Y um eins erhöht (35), so daß er nun auf den nächsten freien Speicherplatz zeigt. Nach jeder Eingabe wird überprüft, ob die RETURN-Taste gedrückt wurde (36): solange dies nicht der Fall ist, wird die Eingabe-Schleife weiter durchlaufen (37).

Wird die Eingabe mit der RETURN-Taste beendet, so wird zu einer Routine verzweigt, die ein RETURN-Zeichen und einen Zeilenvorschub ausgibt (s.u.: 50-53).

```
                        ;AUSGABE
41:    C82F A0 00    OUTPUT   LDY  #$00
42:    C831 B9 87 C8 WRITE    LDA  BUFF.Y
43:    C834 20 D2 FF          JSR  CHROUT
44:    C837 C8                INY
45:    C838 C9 0D             CMP  #CR
46:    C83A D0 F5             BNE  WRITE
47:    C83C 60                RTS
```

Mit dieser Routine kann der soeben eingegebene Text wieder ausgegeben werden.

Das Index-Register Y wird gelöscht, damit es auf den Anfang des Textspeicherbereichs zeigt (41). Das erste Zeichen wird in den Akku geladen (42), dessen Inhalt dann von der ROM-Routine CHROUT ausgegeben wird (43). Y wird inkrementiert, damit es auf das nächste Zeichen zeigt (44). Handelt es sich dabei um ein RETURN-Zeichen, so wird die Ausgabe beendet, ansonsten wird das nächste Zeichen ausgegeben (45-46).

Die folgenden Routinen sind alle sehr einfach: ein Zeichen wird in den Akku geladen und anschließend ausgegeben:

```
                          :RETURN / LINEFEED
50:     C83D A9 0D     CROUT    LDA  #CR
51:     C83F 20 D2 FF           JSR  CHROUT
52:     C842 A9 0A              LDA  #LF
53:     C844 4C D2 FF           JMP  CHROUT
```

Die Routine RETURN / LINEFEED gibt ein RETURN-Zeichen und einen Zeilenvorschub aus.

```
                          :AUSGABE VON LEERZEICHEN
                          :ANZAHL IN X
57:     C847 A9 20     PRBLNK   LDA  #BLANK
58:     C849 20 D2 FF  PRBL1    JSR  CHROUT
59:     C84C CA                 DEX
60:     C84D D0 FA              BNE  PRBL1
61:     C84F 60                 RTS
```

Mit diesem Programm wird die im X-Register stehende Anzahl von Leerzeichen ausgegeben. Selbstverständlich kann hier statt des Leerzeichens auch ein anderes Zeichen angegeben werden.

```
                        ;INVERTIERTE AUSGABE
64:    C850 A9 12    INVOUT   LDA  #18
65:    C852 20 D2 FF          JSR  CHROUT
66:    C855 4C 2F C8          JMP  OUTPUT
```

Diese Routine bewirkt eine invertierte Ausgabe: nachdem der Zeichencode für die invertierte Darstellung ausgegeben wurde, erfolgt eine Verzweigung zu der schon besprochenen Ausgabe-Routine.

```
                        ;KLEINBUCHSTABEN
69:    C858 A9 0E    LOWER    LDA  #14
70:    C85A 4C D2 FF          JMP  CHROUT

                        ;GROSSBUCHSTABEN
73:    C85D A9 8E    UPPER    LDA  #142
74:    C85F 4C D2 FF          JMP  CHROUT
```

Mit dem Aufruf dieser Routinen wird auf Klein- bzw. Großschreibung umgeschaltet.

```
                        ;WARTEN AUF TASTENDRUCK
                        ;ZEICHEN ANSCHLIESSEND IM AKKU
78:    C862 20 E4 FF KBDWT    JSR  GETIN
79:    C865 C9 00             CMP  #$00
80:    C867 F0 F9             BEQ  KBDWT
81:    C869 60                RTS
```

Die Routine KBDWT (Keybord wait) wartet auf einen Tastendruck. Dabei wird die ROM-Routine GETIN verwendet (78), die den ASCII-Wert eines von der Tastatur gelesenen Zeichens in den Akku bringt. Solange keine Taste gedrückt wird, ist dieser Wert gleich Null, und das Programm bleibt in einer Warteschleife (79-80). Nachdem eine Taste gedrückt wurde, erfolgt der Rücksprung mit dem entsprechenden ASCII-Code im Akku (81).

Mit dem folgenden BASIC-Ladeprogramm können die Routinen aus diesem Kapitel in den Speicher gebracht werden. Zuvor sollen noch die dezimalen Anfangsadressen der einzelnen Routinen angegeben werden:

Name	Adresse (hex.)	Adresse (dezimal)
SYSPAR	$C800	51200
INPUT	$C81A	51226
OUTPUT	$C82F	51247
CROUT	$C83D	51261
PRBLNK	$C847	51271
INVOUT	$C850	51280
LOWER	$C858	51288
UPPER	$C85D	51293
KBDWT	$C862	51298
PRBYTE	$C86A	51306
PRADR	$C880	51328

```
1 REM ..... ROM-ROUTINEN
10 DATA  32,253,174, 32,138,173, 32,247,183,165
20 DATA  21,166, 20, 32,128,200, 32,253,174, 32
30 DATA 158,183,138, 76,106,200,169, 62, 32,210
40 DATA 255,160,  0, 32,207,255,153,135,200,200
50 DATA 201, 13,208,245, 76, 61,200,160,  0,185
60 DATA 135,200, 32,210,255,200,201, 13,208,245
70 DATA  96,169, 13, 32,210,255,169, 10, 76,210
80 DATA 255,169, 32, 32,210,255,202,208,250, 96
90 DATA 169, 18, 32,210,255, 76, 47,200,169, 14
100 DATA  76,210,255,169,142, 76,210,255, 32,228
110 DATA 255,201,  0,240,249, 96, 72, 74, 74, 74
120 DATA  74, 32,117,200,104, 41, 15,  9, 48,201
130 DATA  58,144,  2,105,  6, 76,210,255, 32,106
140 DATA 200,138, 76,106,200,  0
150 FOR I = 51200 TO 51335
160 READ K: POKE I,K
170 SUM=SUM+K
180 NEXT I
190 IF SUM <> 18148 THEN PRINT "PRUEFSUMME FALSCH!": END
200 PRINT "PRUEFSUMME RICHTIG!"
```

9 | Kurze Darstellung der in den Programmen nicht verwendeten Befehle

9. Kurze Darstellung der in den Programmen nicht verwendeten Befehle

In den bisher vorgestellten Programmen wurden bereits die meisten Befehle des 6510 benutzt. Einige der noch nicht verwendeten Befehle lassen sich sehr schnell beschreiben, da sie bereits bekannten Befehlen ähnlich sind.

Da wären zunächst zwei weitere Verzweigungsbefehle: BVC (Branch on oVerflow Clear) und BVS (Branch on oVerflow Set), bei denen also die Verzweigung vom Zustand des Übertrag-Flags abhängig ist. BVC verzweigt zu der angegebenen Adresse, wenn das V-Flag (Bit 6 des Status-Registers) gelöscht ist, bei BVS findet die Verzweigung statt, wenn das V-Flag gesetzt ist. Wie bei den anderen Verzweigungsbefehlen ist auch hier nur die relative Adressierung möglich, d.h. das Programm wird an der Stelle fortgesetzt, die sich durch Addition des angegebenen Abstandes (im Zweierkomplement von -128 bis +127) zur Adresse des folgenden Befehls ergibt.

BVC

BVS

Zum Löschen des Übertrag-Flags gibt es den Befehl CLV (CLear oVerflow flag), der also das V-Flag auf den Wert Null setzt.

CLV

Der Vergleichsbefehl CPY (ComPare to register Y) vergleicht den Inhalt der angegebenen Speicherstelle mit dem Index-Register Y. Wie bei den übrigen Vergleichsbefehlen geschieht dies dadurch, daß der Inhalt des adressierten Speichers von dem im Befehl angegebenen Register (hier also Y) subtrahiert wird. Dem Ergebnis entsprechend werden nur die drei Flags N, Z und C aktualisiert: Z = 1, wenn beide Werte gleich sind. N = 1, wenn der Inhalt von Y kleiner ist als der der angegebenen Adresse. C = 1, wenn der Wert von Y größer oder gleich dem Inhalt des angegebenen Speichers ist.

CPY

Der aktuelle Inhalt des Prozessor-Status-Registers kann mit dem Befehl PHP (PusH Processor status on stack) auf den Stack geschoben und entsprechend mit PLP (PuLl Processor status from stack) wieder zurückgeholt werden.

PHP

PLP

Dezimal-Arithmetik

Der 6510-Prozessor kann nicht nur im Binär- sondern auch im Dezimalbetrieb rechnen. Der **SED** Dezimalbetrieb wird eingeschaltet mit dem Befehl SED (SEt Decimal mode). Alle darauf folgenden ADC- und SBC-Befehle werden als **CLD** dezimalarithmetische Operationen durchgeführt. Zum Umschalten in den Binärbetrieb dient der Befehl CLD (CLear Decimal mode).

Beim Rechnen im Dezimalbetrieb wird die BCD-Codierung verwendet. BCD bedeutet: "binary coded decimal", d.h. binär codierte Dezimalziffer. Dabei wird jede der Ziffern von 0 bs 9 durch die entsprechende Binärzahl dargestellt. Um alle 10 Ziffern zu erfassen, werden dazu 4 Bits benötigt. Von den damit möglichen 16 Kombinationen bleiben also 6 ungenutzt. Ein Byte kann somit zwei BCD-Ziffern enthalten, insgesamt also einen Zahlenwert zwischen 0 und 99 darstellen. Ein Übertrag beim Rechnen im Dezimalmodus muß demnach dann erfolgen, wenn die Dezimalzahl 99 überschritten wird.

Programmunterbrechungen (Interrupts)

Eine Programmunterbrechung (Interrupt) ist in gewisser Weise mit einem Unterprogrammaufruf zu vergleichen. Es gibt zwei Möglichkeiten einer solchen Unterbrechung, die durch zwei Steuerleitungen des 6510-Prozessors realisiert werden: IRQ (Interrupt ReQuest) und NMI (Non-Maskable Interrupt).

Sobald eine Unterbrechung stattfindet, wird die Programmausführung mit einer bestimmten Routine (Interrupt-Behandlungs-Routine) fortgesetzt, deren Anfangsadresse in einem fest vereinbarten Speicher steht, der auch Interrupt-Vektor genannt wird. Die Adresse dieses Speichers ist beim maskierten Interrupt (IRQ) $FFFE, $FFFF und beim nicht maskierten Interrupt (NMI) $FFFA, $FFFB. Nachdem diese Routine abgearbeitet wurde, wird das unterbrochene Programm fortgesetzt. Mit dem Wort "maskiert" soll darauf hingewiesen werden, daß das Interrupt durch das Setzen des Interrupt disable Flags I gesteuert werden kann.

Das nicht maskierte Interrupt hat eine höhere Priorität als das maskierte Interrupt, d.h. wenn der Pro-

zessor gerade die Behandlungs-Routine eines nicht maskierten Interrupts bearbeitet, kann dieser Ablauf nicht von einem maskierten Interrupt unterbrochen werden. Umgekehrt wird die Bearbeitung einer Behandlungs-Routine eines maskierten Interrupts durch eine nicht maskierte Interrupt-Routine in jedem Fall unterbrochen.

Um die Behandlung des maskierten Interrupt zu verhindern, kann man das I-Flag setzen. Das nicht maskierte Interrupt läßt sich nicht umgehen, d.h. es wird in jedem Fall behandelt.

Wenn eine Interruptanforderung auftritt, wird zunächst die Bearbeitung des gerade aktuellen Befehls zu Ende geführt. Dann wird festgestellt, über welche der beiden Steuerleitungen die Interruptanforderung erfolgte: kommt die Anforderung über IRQ, so wird das I-Flag überprüft. Eine Unterbrechung findet jedoch nur dann statt, wenn das I-Flag nicht gesetzt ist. Im Gegensatz dazu wird eine Unterbrechung in jedem Fall ausgeführt, wenn sie über die Leitung NMI angefordert wurde.

CLI
SEI

Zum Löschen des Interrupt disable Flag steht der Befehl CLI (CLear Interrupt disable) zur Verfügung. Das Gegenstück zu diesem Befehl ist SEI (SEt Interrupt disable), womit das I-Flag gesetzt wird.

Vor der Ausführung der Interrupt-Routine wird der Inhalt des Programmzählers und des Status-Registers in den Stack gebracht. Für die Zwischenspeicherung des Akkus und der Index-Register ist der Programmierer selbst zuständig.

Um weitere Unterbrechungen zu verhindern, wird vom Prozessor jetzt das I-Flag gesetzt. Danach wird das Programm mit der Interrupt-Routine fortgesetzt, deren Anfangsadresse bei einem IRQ-Interrupt in $FFFE und $FFFF steht und bei einem NMI-Interrupt in $FFFA und $FFFB.

RTI

Entsprechend wie bei einem Unterprogrammaufruf gibt es einen Befehl zur Rückkehr aus einer Unterbrechung - RTI (ReTurn from Interrupt) - mit dem die drei "geretteten" Bytes (Status-Register und Programmzähler) wieder vom Stack zurückgeholt werden, damit das Programm an der Unter-

brechungsstelle wieder fortgesetzt werden kann. Um weitere Programmunterbrechungen zu ermöglichen, wird außerdem das Interrupt-Flag I gelöscht.

Auch der bereits verwendete Befehl BRK ist eine Interrupt-Anweisung: der Inhalt des (um eins erhöhten) Programmzählers sowie der Inhalt des Status-Registers werden in den Stack gebracht. Außerdem wird vorher das Break-Flag B (Bit 4 des Status-Registers) gesetzt. Die Steuerung erfolgt dann über den IRQ-Vektor, d.h. als Startadresse wird der Inhalt der Speicherstellen $FFFE und $FFFF verwendet. Beim Commodore 64 wird bei der Ausführung des BRK-Befehls normalerweise nach BASIC verzweigt, wobei der Bildschirm gelöscht und die READY-Meldung ausgegeben wird.

Abschließend sei der Befehl genannt, der - fast - nichts tut: NOP (No OPeration). NOP führt tatsächlich "keine Operation" durch, braucht aber trotzdem eine gewisse Zeit zur Ausführung, so daß dieser Befehl z.B. zur Erzeugung von Verzögerungsschleifen benutzt werden kann. Außerdem wird NOP als Platzhalter verwendet, wenn ein bestimmter Programmteil getestet werden soll: die Befehle, die von diesem Test ausgenommen werden soll, werden dann durch NOP-Befehle ersetzt.

NOP

10 | Adressierungsarten

10. Adressierungsarten

Die Adressierungsarten des 6510 können in zwei Gruppen eingeteilt werden: indizierte und nicht indizierte Adressierungsarten. - Bei den indizierten Adressierungsarten erhält man die eigentliche Adresse, indem der Inhalt des Index-Registers zu der angegebenen Adresse addiert wird.

10.1 Nicht indizierte Adressierungsarten

In diese Gruppe gehören:

1. implizite Adressierung,
2. unmittelbare Adressierung,
3. absolute Adressierung,
4. Zero Page Adressierung,
5. relative Adressierung.

10.1.1 Implizite Adressierung (Implied Addressing)

Die implizite Adressierung findet bei Ein-Byte-Befehlen statt.

Die folgenden Befehle bewirken Operationen, die ausschließlich im Prozessor ablaufen und benötigen zu ihrer Ausführung nur 2 Taktzyklen: CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA.

Diese Befehle greifen auf den (Stapel-)Speicher zu und benötigen mehr als 2 Taktzyklen: BRK, PHA, PHP, PLA, PLP, RTI, RTS.

10.1.2 Unmittelbare Adressierung (Immediate)

Die Befehle mit unmittelbarer Adressierung sind 2-Byte-Befehle, wobei das zweite Byte eine Konstante enthält.

Die unmittelbare Adressierung benötigt 2 Taktzyklen.

Folgende Befehle ermöglichen eine unmittelbare Adressierung: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC.

10.1.3 Absolute Adressierung (Absolute)

Befehle mit der absoluten Adressierung sind 3-Byte-Befehle. Das zweite Byte enthält das niederwertige Byte, das dritte Byte enthält das höherwertige Byte der Adresse, die die Daten enthält.

Die absolute Adressierung benötigt 4 Taktzyklen mit Ausnahme des JMP-Befehls, der nur 3 Zyklen benötigt.

Befehle mit absoluter Adressierung sind: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

10.1.4 Zero Page Adressierung (Zero Page)

Die Zero Page Adressierung ist ein 2-Byte-Befehl. Das zweite Byte enthält eine 8-Bit-Adresse auf der Zero Page.

Die Zero Page Adressierung benötigt 3 Taktzyklen.

Außer JMP und JSR können alle Befehle mit absoluter Adressierung auch mit der Zero Page Adressierung verwendet werden. Folgende Befehle erlauben die Zero Page Adressierung: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

10.1.5 Relative Adressierung (Relative)

Die relative Adressierung ist ein 2-Byte-Befehl und wird für alle Verzweigungsvorgänge benutzt. Das erste Byte stellt einen Sprungbefehl dar, das zweite gibt in Zweierkomplement-Darstellung an, um wieviel Bytes - vor oder zurück - der Sprung erfolgen soll.

Die Zahl der Taktzyklen hängt vom Ergebnis des zuvor ausgeführten Tests ab: wird ein Verzweigungsbefehl nicht ausgeführt, so benötigen Befehle mit der

relativen Adressierung 2 Bytes. Wird ein Verzweigungsbefehl ohne Seitenüberschreitung ausgeführt, so benötigt ein Befehl 3 Bytes. Erfolgt die Verzweigung über eine Seitengrenze hinaus, so benötigt ein Befehl 4 Bytes.

Nur Verzweigungsbefehle arbeiten mit der relativen Adressierungsart: BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS.

10.2 Indizierte Adressierungsarten

Bei den indizierten Adressierungsarten ist eine Adresse nicht fest vorgegeben, sondern muß noch errechnet werden. Auf diese Weise ist eine sehr flexible Programmierung möglich.

Grundsätzlich kann bei diesen Adressierungsarten zwischen der einfachen indizierten Adressierung und der indirekten Adressierung unterschieden werden:

Die indizierte Adressierung verwendet eine Adresse, die dadurch errechnet wird, daß zu der angegebenen Adresse der Inhalt eines der Index-Register hinzuaddiert wird.

Die indirekte Adressierung verwendet den Inhalt der angegebenen Adresse als Zeiger auf die eigentliche Adresse.

Außerdem sind auch Kombinationen der indizierten und der indirekten Adressierungsart möglich.

Damit ergeben sich insgesamt folgende indizierten Adressierungsarten:

1. absolut indizierte Adressierung,
2. Zero Page Indizierung,
3. indirekte Adressierung,
4. indiziert-indirekte Adressierung,
5. indirekt-indizierte Adressierung.
6. indirekt absolute Adressierung

10.2.1 Absolut indizierte Adressierung

Der 6510 besitzt zwei Index-Register: X und Y. Der Inhalt eines der Index-Register wird zu der angegebenen Adresse addiert. Auf diese Weise kann man z.B. einen zusammenhängenden Speicherbereich durch Inkrementieren eines Index-Registers relativ zur Anfangsadresse dieses Speicherbereiches der Reihe nach ansprechen, um etwa eine Tabelle abzuarbeiten.

Die beiden Index-Register unterscheiden sich dadurch, daß die Indizierung mit X außer der absoluten auch die Zero Page Adressierung erlaubt, wogegen beim Y-Register nur die absolute Adressierung möglich ist.

Die absolut indizierte Adressierung benötigt 4 Taktzyklen ohne bzw. 5 mit Seitenüberschreitung.

Diese Befehle erlauben eine absolute Indizierung mit X: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA.

Diese Befehle erlauben eine absolute Indizierung mit Y: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA.

10.2.2 Zero Page Indizierung

Wie bei der nicht indizierten Zero Page Adressierung ergibt sich auch hier eine Speichereinsparung, da die angegebene Adresse nur ein Byte lang ist.

Bei der Zero Page Indizierung wird ein Taktzyklus mehr als bei den nicht indizierten Befehlen benötigt.

Als Index-Register kann bei folgenden Befehlen nur das X-Register verwendet werden: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, STY. Eine Ausnahme gilt für die Befehle LDX und STX, bei denen die Indizierung mit dem Y-Register erfolgt.

10.2.3 Indirekte Adressierung

Bei dieser Form der Adressierung ist die dem Befehl folgende Adresse die Adresse einer Adresse, d.h. die angegebene Adresse ist ein Zeiger auf die Adresse, die die eigentliche Adresse enthält.

Diese einfache indirekte Adressierung ist nur mit dem JMP-Befehl möglich. Bei den übrigen Befehlen, die eine indirekte Adressierung erlauben, muß die Adressierung zusätzlich indiziert erfolgen.

10.2.4 Indiziert-indirekte Adressierung

Die indiziert-indirekte Adressierung ist ein 2-Byte-Befehl. Bei dieser Adressierungsart wird der Inhalt des X-Registers zu der angegebenen Zero Page Adresse addiert. Das Ergebnis dient als Zeiger auf zwei aufeinanderfolgende Speicherplätze in der Zero Page, die die eigentliche 16-Bit-Adresse enthalten. Auf diese Weise kann man also mit nur einem Byte auf eine 2-Byte-Adresse zugreifen.

Für die indiziert-indirekte Adressierung werden 6 Taktzyklen benötigt.

Folgende Befehle erlauben die indiziert-indirekte Adressierung: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

10.2.5 Indirekt-indizierte Adressierung

Die indirekt-indizierte Adressierung ist ein 2-Byte-Befehl. Bei dieser Adressierungsart erfolgt die Indizierung (im Gegensatz zur indiziert-indirekten) mit dem Y-Register und erst nach der Ermittlung der indirekten Adresse. Die dem Befehl folgende Adresse verweist also auf eine 2-Byte-Adresse auf der Zero Page, zu der der Inhalt des Y-Registers addiert wird, um die eigentliche Adresse zu erhalten.

Diese Adressierungsart ist besonders zur Tabellenverarbeitung geeignet, bei der relativ zum Tabellenanfang auf einen bestimmten Wert zugegriffen werden soll.

Die Ausführungszeit eines indirekt-indizierten Befehls liegt bei 5 bis 6 Taktzyklen.

Folgende Befehle erlauben die indirekt-indizierte Adressierung: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

10.2.6 Indirekt absolute Adressierung

Als einziger Befehl ermöglicht der JMP-Befehl eine absolute (nicht-indizierte) indirekte Adressierung.

Anhang

Alphabetische Kurzübersicht der 6510-Befehle

ADC ADd memory to accumulator with Carry
Mit übertrag addieren

AND "AND" memory with accumulator
Logisches UND

ASL Arithmetic Shift Left one bit (memory or accumulator)
Arithmetisch links schieben

BCC Branch on Carry Clear
Verzweigen bei gelöschtem übertrag

BCS Branch on Carry Set
Verzweigen bei gesetztem übertrag

BEQ Branch on Result EQual to zero
Verzweigen falls Ergebnis gleich Null

BIT test BITs in memory with accumulator
Bits testen

BMI Branch on result MInus
Verzweige falls Ergebnis ~~gleich Null~~ negativ

BNE Branch on result not EQual to zero
Verzweige falls Ergebnis ungleich Null

BPL Branch on result PLus
Verzweige falls Ergebnis positiv

BRK Force BReaK
Unterbrechung durch Software

BVC Branch on oVerflow Clear
Verzweige falls kein überlauf

BVS Branch on oVerflow Set
Verzweige falls überlauf

CLC CLear Carry flag
übertragflagge löschen

CLD CLear Decimal mode
Dezimalmodus abschalten

CLI CLear Interrupt disable bit
Unterbrechungsmaske löschen

CLV CLear oVerflow flag
überlaufsflagge löschen

CMP CoMPare memory and accumulator
Vergleiche Speicher mit AKkumulator

CPX ComPare memory and index X
Vergleiche Speicher mit Register X

CPY ComPare memory and Index Y
Vergleiche Speicher mit Register Y

DEC DECrement memory by one
Speicher dekrementieren

DEX DEcrement index X by one
Register X dekrementieren

DEY DEcrement index Y by one
Register Y dekrementieren

EOR "Exclusive-OR" memory with accumulator
EXCLUSIV-ODER verknüpfen

INC INCrement memory by one
Speicher inkrementieren

INX INcrement index X by one
Register X inkrementieren

INY INcrement index Y by one
Register Y inkrementieren

JMP JuMP to new location
Zur angegebenen Adresse springen

JSR Jump to new location Saving Return address (Jump to Sub-Routine)
Zum Unterprogramm springen

LDA LoaD Accumulator with memory
Akkumulator mit Speicherinhalt laden

LDX LoaD index X with memory
Register X mit Speicherinhalt laden

LDY LoaD index Y with memory
Register Y mit Speicherinhalt laden

LSR Logical Shift Right one bit (memory or accumulator)
Logisch rechts schieben

NOP No OPeration
Keine Operation

ORA "OR" memory with Accumulator
Mit dem Akkumulator ODER-verknüpfen (einschließend)

PHA PusH Accumulator on stack
Akkumulator auf den Stack bringen

PHP PusH Processor status on stack
Statusregister auf den Stack bringen

PLA PuLl Accumulator from stack
Akkumulator vom Stack holen

PLP PuLl Processor status from stack
Statusregister vom Stack holen

ROL ROtate Left one bit (memory or accumulator)
Um ein Bit nach links rotieren

ROR ROtate Right one bit (memory or accumulator)
Um ein Bit nach rechts rotieren

RTI ReTurn from Interrupt
Rückkehr aus einer Programmunterbrechung

RTS ReTurn from Subroutine
Rückkehr aus einem Unterprogramm

SBC SuBtract memory from accumulator with Carry
Mit übertrag subtrahieren

SEC SEt Carry flag
übertragflagge setzen

SED SEt Decimal mode
Dezimalmodus einschalten

SEI SEt Interrupt disable status
Unterbrechungsmaske setzen

STA STore Accumulator in memory
Akkumulatorinhalt im Speicher ablegen

STX STore index X in memory
Register X im Speicher ablegen

STY STore index Y in memory
Register Y im Speicher ablegen

TAX Transfer Accumulator to index X
Akkumulatorinhalt nach Register X übertragen

TAY Transfer Accumulator to index Y
Akkumulator nach Register Y übertragen

TSX Transfer Stack pointer to index X
Stackpointer nach Register X übertragen

TXA Transfer index X to Accumulator
Register X nach Akkumulator übertragen

TXS Transfer index X to Stack pointer
Register X nach Stackpointer übertragen

TYA Transfer index Y to Accumulator
Register Y nach Akkumulator übertragen

Alphabetische übersicht der 6510 - Befehle und Adressierungsarten

In dieser Zusammenfassung werden folgende Notationen verwendet:

A	Akkumulator
X, Y	Index-Register X und Y
M	Memory, Speicher
P	Prozessorstatus-Register
S	Stackpointer
STACK	Stapel-Register
PC	Programm Counter (Programmzähler)
PCH	Programm Counter High (Programmzähler, höherwertiges Byte)
PCL	Programm Counter Low (Programmzähler, niederwertiges Byte)
%	Veränderung (bei Flaggen)
-	Keine Veränderung (bei Flaggen); sonst: Subtraktion
>, <	Transfer

ADC	A + M + C > A,C						
FLAGGE:	N	Z	C	I	D	V	
	%	%	%	-	-	%	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Immediate	ADC #oper		69	2		2	
Zero Page	ADC oper		65	2		3	
Zero Page, X	ADC oper, X		75	2		4	
Absolut	ADC oper		6D	3		4	
Absolut, X	ADC oper, X		7D	3		4*	
Absolut, Y	ADC oper, Y		79	3		4*	
(Indirekt, X)	ADC (oper, X)		61	2		6	
(Indirekt),Y	ADC (oper),Y		71	2		5*	

* +1 Zyklus bei Seitenüberschreitung.

AND	A "AND" M > A						
FLAGGE:	N	Z	C	I	D	V	
	%	%	-	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Immediate	AND #oper		29	2		2	
Zero Page	AND oper		25	2		3	
Zero Page, X	AND oper, X		35	2		4	
Absolut	AND oper		2D	3		4	
Absolut, X	AND oper, X		3D	3		4*	
Absolut, Y	AND oper, Y		39	3		4*	
(Indirekt, X)	AND (oper, X)		21	2		6	
(Indirekt),Y	AND (oper),Y		31	2		5	

* +1 Zyklus bei Seitenüberschreitung.

ASL	C < BIT7<.....<BIT0 < 0

FLAGGE:	N	Z	C	I	D	V
	%	%	%	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Accumulator	ASL A	0A	1	2
Zero Page	ASL oper	06	2	5
Zero Page, X	ASL oper, X	16	2	6
Absolut	ASL oper	0E	3	6
Absolut, X	ASL oper, X	1E	3	7

BCC	Springe falls C=0

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Relative	BCC oper	90	2	2#

+1 Zyklus wenn Verzweigung erfolgt, und
+2 bei Seitenüberschreitung.

BCS	Springe falls C=1						
FLAGGE:	N	Z	C	I	D	V	
	-	-	-	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Relative	BCS oper		B0	2		2*	

* +1 Zyklus wenn Verzweigung erfolgt, und
+2 bei Seitenüberschreitung.

BEQ	Springe falls Z=1						
FLAGGE:	N	Z	C	I	D	V	
	-	-	-	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Relative	BEQ oper		F0	2		2*	

* +1 Zyklus wenn Verzweigung erfolgt, und
+2 bei Seitenüberschreitung.

BIT	A "AND" M , BIT7 > N, BIT6 > V

FLAGGE:	N	Z	C	I	D	V
	BIT7	%	-	-	-	BIT6

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Zero Page	BIT oper	24	2	3
Absolut	BIT oper	2C	3	4

BMI	Springe falls N=0

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Relative	BMI oper	30	2	2*

* +1 Zyklus wenn Verzweigung erfolgt, und
  +2 bei Seitenüberschreitung.

BNE	Springe falls Z=0						
FLAGGE:	N	Z	C	I	D	V	
	-	-	-	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Relative	BNE oper		D0	2		2*	

* +1 Zyklus wenn Verzweigung erfolgt, und
  +2 bei Seitenüberschreitung.

BPL	Springe falls N=0						
FLAGGE:	N	Z	C	I	D	V	
	-	-	-	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Relative	BPL oper		10	2		2*	

* +1 Zyklus wenn Verzweigung erfolgt, und
  +2 bei Seitenüberschreitung.

BRK	PC + 2 > STACK P - STACK (FFFE,FFFF) > PC					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	1	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	BRK	00	1	7

Ein BRK-Befehl kann nicht durch Setzen von I = 1 maskiert werden

BVC	Springe falls V=0					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Relative	BCC oper	50	2	2*

* +1 Zyklus wenn Verzweigung erfolgt, und
+2 bei Seitenüberschreitung.

BVS	Springe falls V=1					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Relative	BVS oper	70	2	2*

* +1 Zyklus wenn Verzweigung erfolgt, und
  +2 bei Seitenüberschreitung.

CLC	0 > C					
FLAGGE:	N	Z	C	I	D	V
	-	-	0	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	CLC	18	1	2

CLD	0 > D					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	0	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	CLD	D8	1	2

CLI	0 > I					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	0	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	CLI	58	1	2

CLV	0 > V

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	0

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	CLV	B8	1	2

CMP	A - M

FLAGGE:	N	Z	C	I	D	V
	%	%	%	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Immediate	CMP #oper	C9	2	2
Zero Page	CMP oper	C5	2	3
Zero Page, X	CMP oper, X	D5	2	4
Absolut	CMP oper	CD	3	4
Absolut, X	CMP oper, X	DD	3	4*
Absolut, Y	CMP oper, Y	D9	3	4*
(Indirekt, X)	CMP (oper, X)	C1	2	6
(Indirekt),Y	CMP (oper),Y	D1	2	5*

* +1 Zyklus bei Seitenüberschreitung.

CPX	X - M						
FLAGGE:	N	Z	C	I	D	V	
	%	%	%	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Immediate	CPX #oper		E0	2		2	
Zero Page	CPX oper		E4	2		3	
Absolut	CPX oper		EC	3		4	

CPY	Y - M						
FLAGGE:	N	Z	C	I	D	V	
	%	%	%	-	-	-	
Adressierungs-art	Mnemonik		Op-code	Byte-anzahl		Zyklen	
Immediate	CPY #oper		C0	2		2	
Zero Page	CPY oper		C4	2		3	
Absolut	CPY oper		CC	3		4	

DEC	M - 1 > M

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Zero Page	DEC oper	C6	2	5
Zero Page, X	DEC oper, X	D6	2	6
Absolut	DEC oper	CE	3	6
Absolut, X	DEC oper, X	DE	3	7

DEX	X - 1 > X

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	DEX	CA	1	2

DEY	Y -1 > Y

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	DEY	88	1	2

EOR	A "EOR" M > A

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Immediate	EOR #oper	49	2	2
Zero Page	EOR oper	45	2	3
Zero Page, X	EOR oper, X	55	2	4
Absolut	EOR oper	4D	3	4
Absolut, X	EOR oper, X	5D	3	4*
Absolut, Y	EOR oper, Y	59	3	4*
(Indirekt, X)	EOR (oper, X)	41	2	6
(Indirekt),Y	EOR (oper),Y	51	2	5*

* +1 Zyklus bei Seitenüberschreitung.

INC	M + 1 > M

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Zero Page	INC oper	E6	2	5
Zero Page, X	INC oper, X	F6	2	6
Absolut	INC oper	EE	3	6
Absolut, X	INC oper, X	FE	3	7

INX	X + 1 > X

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	INX	E8	1	2

INY	Y + 1 > Y

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	INY	C8	1	2

JMP	(PC + 1) > PCL (PC + 2) > PCH

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Absolut	JMP oper	4C	3	3
Indirekt	JMP (oper)	6C	3	5

JSR	PC + 2) STACK (PC + 1)) PCL (PC + 2)) PCH					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Absolut	JSR oper	20	3	6

LDA	M) A					
FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Immediate	LDA #oper	A9	2	2
Zero Page	LDA oper	A5	2	3
Zero Page, X	LDA oper, X	B5	2	4
Absolut	LDA oper	AD	3	4
Absolut, X	LDA oper, X	BD	3	4*
Absolut, Y	LDA oper, Y	B9	3	4*
(Indirekt, X)	LDA (oper, X)	A1	2	6
(Indirekt),Y	LDA (oper),Y	B1	2	5*

* +1 Zyklus bei Seitenüberschreitung.

LDX	M > X

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Immediate	LDX #oper	A2	2	2
Zero Page	LDX oper	A6	2	3
Zero Page, Y	LDX oper, Y	B6	2	4
Absolut	LDX oper	AE	3	4
Absolut, Y	LDX oper, Y	BE	3	4*

* +1 Zyklus bei Seitenüberschreitung.

LDY	M > Y

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Immediate	LDY #oper	A0	2	2
Zero Page	LDY oper	A4	2	3
Zero Page, X	LDY oper, X	B4	2	4
Absolut	LDY oper	AC	3	4
Absolut, X	LDY oper, X	BC	3	4*

* +1 Zyklus bei Seitenüberschreitung.

LSR	0 > BIT7>.....>BIT0 > C M ODER A					
FLAGGE:	N	Z	C	I	D	V
	0	%	%	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Akkumulator	LSR A	4A	1	2
Zero Page	LSR oper	46	2	5
Zero Page, X	LSR oper, X	56	2	6
Absolut	LSR oper	4E	3	6
Absolut, X	LSR oper, X	5E	3	7

NOP	keine Operation					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	NOP	EA	1	2

ORA	A "OR" M > A

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Immediate	ORA #oper	09	2	2
Zero Page	ORA oper	05	2	3
Zero Page, X	ORA oper, X	15	2	4
Absolut	ORA oper	0D	3	4
Absolut, X	ORA oper, X	1D	3	4*
Absolut, Y	ORA oper, Y	19	3	4*
(Indirekt, X)	ORA (oper, X)	01	2	6
(Indirekt),Y	ORA (oper),Y	11	2	5

* +1 Zyklus bei Seitenüberschreitung.

PHA	A > STACK

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	PHA	48	1	3

PHP	P > STACK					
FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	PHP	08	1	3

PLA	STACK > A					
FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	PLA	68	1	4

PLP	STACK > P									
FLAGGE:	N	Z	C	I	D	V				
	<---------VON STACK---------->									
Adressierungs-art	Mnemonik			Op-code		Byte-anzahl		Zyklen		
Implizit	PLP			28		1		4		

ROL	>---------->									
	< BIT7<.....<BIT0 < C <									
	M ODER A									
FLAGGE:	N	Z	C	I	D	V				
	%	%	%	-	-	-				
Adressierungs-art	Mnemonik			Op-code		Byte-anzahl		Zyklen		
Akkumulator	ROL	A		2A		1		2		
Zero Page	ROL	oper		26		2		5		
Zero Page, X	ROL	oper, X		36		2		6		
Absolut	ROL	oper		2E		3		6		
Absolut, X	ROL	oper, X		3E		3		7		

ROR	<---------- > C > BIT7<.....<BIT0 < M ODER A
FLAGGE:	N Z C I D V % % % - - -

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Akkumulator	ROR A	6A	1	2
Zero Page	ROR oper	66	2	5
Zero Page, X	ROR oper, X	76	2	6
Absolut	ROR oper	6E	3	6
Absolut, X	ROR oper, X	7E	3	7

RTI	STACK > P STACK > PC
FLAGGE:	N Z C I D V <---------VON STACK---------->

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	RTI	40	1	6

RTS	STACK > PC PC + 1 > PC

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	RTS	60	1	6

SBC	A - M - C > A

FLAGGE:	N	Z	C	I	D	V
	%	%	%	-	-	%

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Immediate	SBC #oper	E9	2	2
Zero Page	SBC oper	E5	2	3
Zero Page, X	SBC oper, X	F5	2	4
Absolut	SBC oper	ED	3	4
Absolut, X	SBC oper, X	FD	3	4*
Absolut, Y	SBC oper, Y	F9	3	4*
(Indirekt, X)	SBC (oper, X)	E1	2	6
(Indirekt),Y	SBC (oper),Y	F1	2	5*

* +1 Zyklus bei Seitenüberschreitung.

SEC	1 > C						
FLAGGE:	N	Z	C	I	D	V	
	-	-	1	-	-	-	
Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen			
Implizit	SEC	38	1	2			

SED	1 > D						
FLAGGE:	N	Z	C	I	D	V	
	-	-	-	-	1	-	
Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen			
Implizit	SED	F8	1	2			

SEI	1 > I

FLAGGE:	N	Z	C	I	D	V
	-	-	-	1	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	SEI	78	1	2

STA	A > M

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Zero Page	STA oper	85	2	3
Zero Page, X	STA oper, X	95	2	4
Absolut	STA oper	8D	3	4
Absolut, X	STA oper, X	9D	3	5
Absolut, Y	STA oper, Y	99	3	5
(Indirekt, X)	STA (oper, X)	81	2	6
(Indirekt),Y	STA (oper),Y	91	2	6

STX	X > M

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Zero Page	STX oper	86	2	3
Zero Page, Y	STX oper, Y	96	2	4
Absolut	STX oper	8E	3	4

STY	Y > M

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Zero Page	STY oper	84	2	3
Zero Page, X	STY oper, X	94	2	4
Absolut	STY oper	8C	3	4

TAX	A > X					
FLAGGE:	N %	Z %	C -	I -	D -	V -

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	TAX	AA	1	2

TAY	A > Y					
FLAGGE:	N %	Z %	C -	I -	D -	V -

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	TAY	A8	1	2

TSX	S > X					
FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	TSX	BA	1	2

TXA	X > A					
FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungsart	Mnemonik	Opcode	Byteanzahl	Zyklen
Implizit	TXA	8A	1	2

TXS	X > S

FLAGGE:	N	Z	C	I	D	V
	-	-	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	TXS	9A	1	2

TYA	Y > A

FLAGGE:	N	Z	C	I	D	V
	%	%	-	-	-	-

Adressierungs-art	Mnemonik	Op-code	Byte-anzahl	Zyklen
Implizit	TYA	98	1	2

6510-Befehlsliste in hexadezimaler Reihenfolge

00 - BRK
01 - ORA - (indirekt,X)
02 - *
03 - *
04 - *
05 - ORA - Zero Page
06 - ASL - Zero Page
07 - *
08 - PHP
09 - ORA - unmittelbar
0A - ASL - Akkumulator
0B - *
0C - *
0D - ORA - absolut
0E - ASL - absolut
0F - *
10 - BPL
11 - ORA - (indirekt),Y
12 - *
13 - *
14 - *
15 - ORA - Zero Page,X
16 - ASL - Zero Page,X
17 - *
18 - CLC
19 - ORA - absolut,Y
1A - *
1B - *
1C - *
1D - ORA - absolut,X
1E - ASL - absolut,X
1F - *

20 - JSR
21 - AND - (indirekt,X)
22 - *
23 - *
24 - BIT - Zero Page
25 - AND - Zero Page
26 - ROL - Zero Page
27 - *
28 - PLP
29 - AND - unmittelbar
2A - ROL - Akkumulator
2B - *
2C - BIT - absolut
2D - AND - absolut
2E - ROL - absolut
2F - *
30 - BMI
31 - AND - (indirekt),Y
32 - *
33 - *
34 - *
35 - AND - Zero Page,X
36 - ROL - Zero Page,X
37 - *
38 - SEC
39 - AND - absolut,Y
3A - *
3B - *
3C - *
3D - AND - absolut,X
3E - ROL - absolut,X
3F - *

40 - RTI
41 - EOR - (indirekt,X)
42 - *
43 - *
44 - *
45 - EOR - Zero Page
46 - LSR - Zero Page
47 - *
48 - PHA
49 - EOR - unmittelbar
4A - LSR - Akkumulator
4B - *
4C - JMP - absolut
4D - EOR - absolut
4E - LSR - absolut
4F - *
50 - BVC
51 - EOR - (indirekt),Y
52 - *
53 - *
54 - *
55 - EOR - Zero Page,X
56 - LSR - Zero Page,X
57 - *
58 - CLI
59 - EOR - absolut,Y
5A - *
5B - *
5C - *
5D - EOR - absolut,X
5E - LSR - absolut,X
5F - *

60 - RTS
61 - ADC - (indirekt,X)
62 - *
63 - *
64 - *
65 - ADC - Zero Page
66 - ROR - Zero Page
67 - *
68 - PLA
69 - ADC - unmittelbar
6A - ROR - Akkumulator
6B - *
6C - JMP - indirekt
6D - ADC - absolut
6E - ROR - absolut
6F - *
70 - BVS
71 - ADC - (indirekt),Y
72 - *
73 - *
74 - *
75 - ADC - Zero Page,X
76 - ROR - Zero Page,X
77 - *
78 - SEI
79 - ADC - absolut,Y
7A - *
7B - *
7C - *
7D - ADC - absolut,X
7E - ROR - absolut,X
7F - *

```
80 - *                          A0 - LDY - unmittelbar
81 - STA - (indirekt,X)         A1 - LDA - (indirekt,X)
82 - *                          A2 - LDX - unmittelbar
83 - *                          A3 - *
84 - STY - Zero Page            A4 - LDY - Zero Page
85 - STA - Zero Page            A5 - LDA - Zero Page
86 - STX - Zero Page            A6 - LDX - Zero Page
87 - *                          A7 - *
88 - DEY                        A8 - TAY
89 - *                          A9 - LDA - unmittelbar
8A - TXA                        AA - TAX
8B - *                          AB - *
8C - STY - absolut              AC - LDY - absolut
8D - STA - absolut              AD - LDA - absolut
8E - STX - absolut              AE - LDX - absolut
8F - *                          AF - *
90 - BCC                        B0 - BCS
91 - STA - (indirekt),Y         B1 - LDA - (indirekt),Y
92 - *                          B2 - *
93 - *                          B3 - *
94 - STY - Zero Page,X          B4 - LDY - Zero Page,X
95 - STA - Zero Page,X          B5 - LDA - Zero Page,X
96 - STX - Zero Page,Y          B6 - LDX - Zero Page,Y
97 - *                          B7 - *
98 - TYA                        B8 - CLV
99 - STA - absolut,Y            B9 - LDA - absolut,Y
9A - TXS                        BA - TSX
9B - *                          BB - *
9C - *                          BC - LDY - absolut,X
9D - STA - absolut,X            BD - LDA - absolut,X
9E - *                          BE - LDX - absolut,Y
9F - *                          BF - *
```

C0 - CPY - unmittelbar
C1 - CMP - (indirekt,X)
C2 - *
C3 - *
C4 - CPY - Zero Page
C5 - CMP - Zero Page
C6 - DEC - Zero Page
C7 - *
C8 - INY
C9 - CMP - unmittelbar
CA - DEX
CB - *
CC - CPY - absolut
CD - CMP - absolut
CE - DEC - absolut
CF - *
D0 - BNE
D1 - CMP - (indirekt),Y
D2 - *
D3 - *
D4 - *
D5 - CMP - Zero Page,X
D6 - DEC - Zero Page,X
D7 - *
D8 - CLD
D9 - CMP - absolut,Y
DA - *
DB - *
DC - *
DD - CMP - absolut,X
DE - DEC - absolut,X
DF - *

E0 - CPX - unmittelbar
E1 - SBC - (indirekt,X)
E2 - *
E3 - *
E4 - CPX - Zero Page
E5 - SBC - Zero Page
E6 - INC - Zero Page
E7 - *
E8 - INX
E9 - SBC - unmittelbar
EA - NOP
EB - *
EC - CPX - absolut
ED - SBC - absolut
EE - INC - absolut
EF - *
F0 - BEQ
F1 - SBC - (indirekt),Y
F2 - *
F3 - *
F4 - *
F5 - SBC - Zero Page,X
F6 - INC - Zero Page,X
F7 - *
F8 - SED
F9 - SBC - absolut,Y
FA - *
FB - *
FC - *
FD - SBC - absolut,X
FE - INC - absolut,X
FF - *

Klassifikation der 6510-Befehle

1. Transferbefehle

LDA LDX LDY PHA PHP TAX TAY TSX
STA STX STY PLA PLP TXA TYA TXS

2. Verarbeitungsbefehle

2.1 Arithmetische Operationen

ADC SBC

2.2 Logische Operationen

AND EOR ORA BIT

2.3 Schiebeoperationen

ASL LSR ROL ROR

2.4 Inkrementieren und Dekrementieren

INC INX INY
DEC DEX DEY

3. Vergleichs- und Verzweigungsbefehle

CMP CPX CPY

BCC BEQ BMI BVC
BCS BNE BPL BVS

JMP JSR
RTS RTI

4. Steuerbefehle

CLC CLD CLI CLV BRK NOP
SEC SED SEI

Hexadezimalcode-Eingabe / Disassembler-Programm

Mit dem folgenden BASIC-Programm können Maschinenprogramme eingegeben werden, die in hexadezimaler Form vorliegen. Dazu wird der Hexadezimalcode in die DATA-Zeilen geschrieben, die sich am Ende des Programms befinden. Die Hexadezimalcodes werden vom Programm in Dezimalwerte umgerechnet und in den gewünschten Speicherbereich gebracht.

Weiterhin verfügt dieses Programm über die Möglichkeit, ein disassembliertes Listing des Maschinenprogramms auf dem Bildschirm auszugeben, so daß Sie vor der endgültigen Eingabe in den Speicher noch einmal das Listing in Assemblerschreibweise überprüfen können.

Mit dem eingebauten Disassembler können Sie darüber hinaus auch Programme disassemblieren, die sich bereits im Speicher des Computers befinden. Auf diese Weise können Sie auch auf Entdeckungsreise gehen und sich die ROM-Routinen des Commodore 64 ansehen.

Nach dem Programmstart werden in einem Menue die folgenden Möglichkeiten angeboten:

1 DATEN EINLESEN
2 HEXCODE LISTEN
3 DISASSEMBLIEREN
4 MASCHINEN-PROG. - RAM
0 ENDE

Wählen Sie die Option DATEN EINLESEN, so fragt das Programm, ob die Daten aus DATA-Zeilen oder aus dem RAM- (bzw. ROM-) Bereich des Computers gelesen werden sollen.

Beim Einlesen aus dem RAM- bzw. (ROM-) Bereich müssen Sie anschließend die Anfangs- und Endadreesse des betreffenden Speicherbereiches angeben. Diese Angaben können sowohl im Dezimal- als auch im Hexadezimalsystem erfolgen: beginnt die Adresse mit einem $-Zeichen, so wird sie als Hexadezimalzahl interpretiert, sonst als Dezimalzahl.

Beim Einlesen aus den DATA-Zeilen entfallen diese Angaben natürlich. Dafür wird in diesem Fall automa-

tisch $C000 (49152 dezimal) als Standard-Anfangsadresse für das anschließende Speichern bestimmt. Selbstverständlich kann diese Adresse vor dem tatsächlichen Speichern (4. Menue-Punkt) noch geändert werden. Allerdings sollten Sie sich in dem Fall vorher davon überzeugen, daß Sie keinen unzulässigen Speicherbereich angeben, da Ihr Computer sich sonst sehr leicht von Ihnen verabschieden könnte und Ihr Programm zerstört würde.

Mit dem 2. Menue-Punkt HEXCODE LISTEN können Sie sich die Werte aus den DATA-Zeilen zur Kontrolle auch noch einmal in der hexadezimalen Darstellung anschauen.

Mit der Wahl von Punkt 3 können Sie das gewünschte Maschinenprogramm DISASSEMBLIEREN. Sie brauchen dabei keine Angst zu haben, daß Ihnen das Listing auf dem Bildschirm davonläuft, da das Auflisten nach jeder gefüllten Bildschirmseite erst auf Tastendruck fortgesetzt wird.

Wählen Sie aus dem Menue den Punkt 4 (MASCHINEN-PROGR. - RAM), so wird das vorher aus den DATA-Zeilen eingelesene Programm in den RAM-Speicher gebracht. Dabei können Sie (wie schon erwähnt) die vorgegebene Anfangsadresse übernehmen oder auch einen neuen Wert angeben.

```
10 REM ............................ HEXCODE-EINGABE/DISASSEMBLER
20 PRINT CHR$(5):REM ZEICHENFARBE WEISS
30 PRINT"■":PRINT TAB(5)"■ HEXCODE-EINGABE/DISASSEMBLER ":PRINT
40 REM ........................ EINLESEN DER HEXADEZIMAL-ZEICHEN
50 PRINT:PRINT" BITTE WARTEN ";
60 DIM HD$(15): REM HEXADEZIMAL-ZEICHEN
70 DIM H$(1000):REM HEXADEZIMAL-CODES
80 FOR I=0 TO 15
90 READ HD$(I)
100 NEXT I
110 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
120 REM ..............................EINLESEN DER 6510-BEFEHLE
130 DIM B$(15,15,2)
140 FOR I=0 TO 15
150 PRINT".";
160 FOR J=0 TO 14
170 FOR K=0 TO 2
180 READ B$(I,J,K)
190 NEXT K
200 NEXT J
210 NEXT I
220 GOTO 860
230 REM .......... HEXADEZIMAL --> DEZIMAL (1 BYTE)
240 L$=LEFT$(H$,1): R$=RIGHT$(H$,1)
250 IF ASC(R$)<65 THEN D=VAL(R$): GOTO 270
260 D=ASC(R$)-55
270 IF ASC(L$)<65 THEN D=D+16*VAL(L$): GOTO 290
280 D=D+16*(ASC(L$)-55)
290 RETURN
300 REM .......... HEXADEZIMAL --> DEZIMAL (ALLG.)
310 L=LEN(H$): D=0: F$=""
320 FOR X=1 TO L
330 M$=MID$(H$,X,1)
340 IF M$>="0" AND M$<="F" THEN 360
350 PRINT" FEHLER: ";H$;" IST KEIN HEXCODE!":F$="F"
360 IF M$<"A" THEN D=D+VAL(M$)*16↑(L-X):GOTO 380
370 D=D+(ASC(M$)-55)*16↑(L-X)
380 NEXT X
390 RETURN
```

```
400 REM ..... DEZIMAL --> HEXADEZIMAL
410 H$="": DEZ=D
420 Q=INT(DEZ/16)
430 R=(DEZ/16-Q)*16
440 H$=HD$(R)+H$
450 DEZ=Q
460 IF Q>0 THEN GOTO 420
470 IF LEN(H$)<2 THEN H$="0"+H$
480 RETURN
490 REM .....EINLESEN DER DATEN
500 PRINT"▙":PRINT" DATEN AUS D)ATA-ZEILEN ODER AUS":PRINT
510 PRINT TAB(11)"R)AM-SPEICHER LESEN";
520 INPUT DR$:PRINT
530 IF DR$= "D" THEN 760
540 IF DR$<>"R" THEN 520
550 REM ..... EINLESEN AUS RAM-SPEICHER
560 PRINT:PRINT" ADRESSEN KOENNEN DEZIMAL ODER HEXA-"
570 PRINT:PRINT" DEZIMAL (MIT '$') EINGEGEBEN WERDEN":PRINT
580 PRINT:INPUT" ANFANGSADRESSE";A1$
590 IF LEFT$(A1$,1)="$" THEN A1$=RIGHT$(A1$,LEN(A1$)-1):GOTO 610
600 A1=VAL(A1$):D=A1:GOSUB 400:A1$=H$:GOTO 620
610 H$=A1$:GOSUB 300:A1=D:IF F$="F" THEN 580
620 INPUT" ENDADRESSE    ";A2$
630 IF LEFT$(A2$,1)="$" THEN A2$=RIGHT$(A2$,LEN(A2$)-1):GOTO 650
640 A2=VAL(A2$):D=A2:GOSUB 400:A2$=H$:GOTO 660
650 H$=A2$:GOSUB 300:A2=D:IF F$="F" THEN 620
660 N=A2-A1+1
670 IF A2<A1 THEN PRINT" ANFANGSADRESSE > ENDADRESSE!":GOTO 580
680 PRINT:PRINT" DIE DATEN WERDEN EINGELESEN ";
690 FOR I=A1 TO A2
700 PRINT".";
710 D=PEEK(I)
720 GOSUB 400
730 H$(I-A1+1)=H$
740 NEXT I:PRINT:PRINT
750 RETURN
```

```
760 REM ..... EINLESEN DER HEXADEZIMAL-CODES AUS DATA-ZEILEN
770 PRINT:PRINT" DIE DATEN WERDEN EINGELESEN ";
780 A1$="C000":A1=49152:REM STANDARD-ANFANGSADRESSE
790 I=1
800 PRINT".";
810 READ H$(I)
820 IF H$(I)<>"$" THEN I=I+1: GOTO 800
830 N=I-1:A2=A1+N-1:D=A2:GOSUB 400:A2$=H$
840 PRINT
850 RETURN
860 REM ..... MENUE
870 PRINT"□"
880 PRINT"      HEXCODE-EINGABE/DISASSEMBLER      "
890 IF N=0 THEN PRINT:PRINT: GOTO 940
900 A=A1
910 PRINT TAB(6)"ANFANGSADRESSE: $";A1$;" (";A1;")
920 PRINT TAB(6)"ENDADRESSE:     $";A2$;" (";A2;")"
930 PRINT TAB(6)"LAENGE:          ";N;"BYTES":PRINT:PRINT
940 PRINT TAB(10)"1  DATEN EINLESEN":PRINT
950 PRINT TAB(10)"2  HEXCODE LISTEN":PRINT
960 PRINT TAB(10)"3  DISASSEMBLIEREN":PRINT
970 PRINT TAB(10)"4  MASCHINEN-PROG. -> RAM":PRINT
980 PRINT TAB(10)"0  ENDE"
990 POKE 214,19: POKE 211,5: SYS 58732: INPUT"-->";A$
1000 IF A$="0" THEN END
1010 IF N=0 THEN IF A$<>"1" THEN 990
1020 IF (A$<"1" OR A$>"4") THEN 990
1030 ON VAL(A$) GOSUB 490,1070,1160,1700
1040 POKE 214,23: SYS 58732:PRINT"(MENUE: RETURN-TASTE DRUECKEN) ";
1050 GET A$: IF A$="" THEN 1050
1060 GOTO 860
1070 REM ..... HEXCODE LISTEN
1080 PRINT"□":PRINT" ADRESSE    HEXADEZIMAL-CODE   ■": PRINT
1090 D=A1:GOSUB 400:PRINT" ";H$;"  ";
1100 FOR I= 1 TO N
1110 PRINT H$(I);" ";
1120 IF I/8=INT(I/8) THEN D=D+8:GOSUB 400:PRINT:PRINT" ";H$;"  ";
1130 NEXT I
1140 PRINT
1150 RETURN
```

```
1160 REM ..................................... DISASSEMBLIEREN
1170 PRINT"␣"
1180 PRINT" ADRESSE  HEX-CODE       ASSEMBLER-CODE "
1190 I=1: REM INDEX FUER H$(#)
1200 Z=1: REM ZEILENZAEHLER FUER BILDSCHIRMAUSGABE
1210 T0=10:T1=25:T2=30: REM TABULATORPOSITIONEN
1220 IF LEN(H$(I))=1 THEN H$(I)="0"+H$(I)
1230 REMü..... HEXCODE --> FELDINDIZES FUER B$(#,#,#)
1240 L$=LEFT$(H$(I),1):R$=RIGHT$(H$(I),1)
1250 IF L$)="0" AND L$(="9" THEN I1=VAL(L$): GOTO 1280
1260 IF L$)="A" AND L$(="F" THEN I1=ASC(L$)-55: GOTO 1280
1270 PRINT" KEIN HEXCODE: ";H$(I);" (BYTE-NR.:";I:GOTO 1690
1280 IF R$)="0" AND R$(="9" THEN I2=VAL(R$): GOTO 1310
1290 IF R$)="A" AND R$(="F" THEN I2=ASC(R$)-55: GOTO 1310
1300 PRINT" KEIN HEXCODE: ";H$(I);" (BYTE-NR.:";I;")":GOTO 1690
1310 REM ..... UNGUELTIGER HEXCODE?
1320 IF I2=15 THEN 1340
1330 IF B$(I1,I2,0)<>"-" THEN 1360
1340 D=A:GOSUB 400:PRINT" ";H$;TAB(T0)H$(I);TAB(T1)"???"
1350 A=A+1:I=I+1: GOTO 1640
1360 REM ............... 1-BYTE-BEFEHLE:
1370 IF B$(I1,I2,1)<>"1" THEN 1400
1380 D=A:GOSUB 400:PRINT" ";H$;TAB(T0)H$(I);TAB(T1)B$(I1,I2,0)
1390 A=A+1: GOTO 1640
1400 REM AUSGABE: ADR., HEX-CODE, BEFEHL
1410 BY=VAL(B$(I1,I2,1))
1420 D=A:GOSUB 400:PRINT" ";H$;:A=A+BY:PRINT TAB(T0);
1430 FOR J=0 TO BY-1
1440 PRINT H$(I+J);"■";
1450 NEXT J
1460 PRINT TAB(T1);B$(I1,I2,0);
1470 REM ............... 2-BYTE-BEFEHLE
1480 IF B$(I1,I2,1) = "3" THEN GOTO 1590
1490 IF B$(I1,I2,2) = "-" THEN PRINT TAB(T2);"$";H$(I+1):GOTO 1640
1500 IF B$(I1,I2,2) = "#" THEN PRINT TAB(T2);"#$";H$(I+1):GOTO 1640
1510 IF B$(I1,I2,2) = "(X)" THEN PRINT TAB(T2);"($";H$(I+1);",X)":GOTO 1640
1520 IF B$(I1,I2,2) = "()Y" THEN PRINT TAB(T2);"($";H$(I+1);"),Y":GOTO 1640
1530 IF B$(I1,I2,2) = "X" THEN PRINT TAB(T2);"$";H$(I+1);",X":GOTO 1640
1540 IF B$(I1,I2,2) = "Y" THEN PRINT TAB(T2);"$";H$(I+1);",Y":GOTO 1640
1550 IF B$(I1,I2,2) <> "R" THEN GOTO 1590
```

```
1560 H$=H$(I+1): GOSUB 230
1570 IF D<128 THEN PRINT " ■";:D=A+D:GOSUB 400:PRINT" ■ $"H$;" ":GOTO 1640
1580 IF D>=128 THEN D=A-1-(255-D):GOSUB 400:PRINT" ■ $"H$;" ":GOTO 1640
1590 REM ................. 3-BYTE-BEFEHLE
1600 IF B$(I1,I2,2)="-" THEN PRINT TAB(T2)"$";H$(I+2);H$(I+1):GOTO 1640
1610 IF B$(I1,I2,2)="X" THEN PRINT TAB(T2)"$";H$(I+2);H$(I+1);",X":GOTO 1640
1620 IF B$(I1,I2,2)="Y" THEN PRINT TAB(T2)"$";H$(I+2);H$(I+1);",Y":GOTO 1640
1630 IF B$(I1,I2,2)="()" THEN PRINT TAB(T2)"($";H$(I+2);H$(I+1);")"
1640 IF Z/20<>INT(Z/20) THEN GOTO 1680
1650 PRINT: PRINT"<WEITER: RETURN-TASTE DRUECKEN >";
1660 GET A$: IF A$="" THEN 1660
1670 PRINT"■"
1680 IF I+VAL(B$(I1,I2,1)) <= N THEN I=I+VAL(B$(I1,I2,1)): Z=Z+1: GOTO 1220
1690 RETURN
1700 REM ...... MASCHINENPROGRAMM -> RAM-SPEICHER
1710 PRINT"■":PRINT TAB(6)"■ MASCHINEN-PROGRAMM --> RAM ":PRINT
1720 PRINT:PRINT" ANFANGSADR.: (RETURN=$";A1$;" /";STR$(A1);")";
1730 A$="":INPUT A$
1740 IF A$="" THEN 1780
1750 IF LEFT$(A$,1)="$" THEN A$=RIGHT$(A$,LEN(A$)-1):GOTO 1770
1760 A1=VAL(A$):D=A1:GOSUB 400:A1$=H$:GOTO 1780
1770 A1$=A$:H$=A1$:GOSUB 300:A1=D:IF F$="F" THEN 1720
1780 IF A1$>="C000" AND A1$<="CFFF" THEN 1820
1790 PRINT" ANFANGSADRESSE: $";A1$;"(";A1;")":PRINT
1800 PRINT"■ WARNUNG!!! ADRESSE LIEGT NICHT IM   "
1810 PRINT"■ NORMALEN RAM-BEREICH ($C000-$CFFF)! ":PRINT
1820 INPUT" SIND SIE SICHER (J/N)";F$:PRINT
1830 IF F$<>"J" THEN 1710
1840 A2=A1+N-1:D=A2:GOSUB 400:A2$=H$
1850 PRINT" ENDADRESSE:      $";A2$;" (";A2;")":PRINT
1860 F$="":INPUT" ADRESSEN RICHTIG (J/N)";F$
1870 IF F$<>"J" THEN 1700
1880 PRINT:PRINT"■ ADRESSE  DEZ. HEX.■":PRINT
1890 FOR I=1 TO N
1900 H$=H$(I)
1910 GOSUB 230
1920 PRINT A1+I-1;TAB(9)D;TAB(15)H$
1930 POKE A1+I-1,D
1940 NEXT I:PRINT:PRINT
1950 RETURN
```

```
1960 REM ..... 6510-BEFEHL, BYTE-ZAHL, ADRESSIERUNGS-ART
1970 DATA BRK,1,-,ORA,2,(X),-,-,-,-,-,-,-,-,-,ORA,2,-,ASL,2,-,-,-,-
1980 DATA PHP,1,-,ORA,2,#,ASL,1,-,-,-,-,-,-,-,ORA,3,-,ASL,3,-
1990 DATA BPL,2,R,ORA,2,()Y,-,-,-,-,-,-,-,-,-,ORA,2,X,ASL,2,X,-,-,-
2000 DATA CLC,1,-,ORA,3,Y,-,-,-,-,-,-,-,-,-,ORA,3,X,ASL,3,X
2010 DATA JSR,3,-,AND,2,(X),-,-,-,-,-,-,BIT,2,-,AND,2,-,ROL,2,-,-,-,-
2020 DATA PLP,1,-,AND,2,#,ROL,1,-,-,-,-,BIT,3,-,AND,3,-,ROL,3,-
2030 DATA BMI,2,R,AND,2,()Y,-,-,-,-,-,-,-,-,-,AND,2,X,ROL,2,X,-,-,-
2040 DATA SEC,1,-,AND,3,Y,-,-,-,-,-,-,-,-,-,AND,3,X,ROL,3,X
2050 DATA RTI,1,-,EOR,2,(X),-,-,-,-,-,-,-,-,-,EOR,2,-,LSR,2,-,-,-,-
2060 DATA PHA,1,-,EOR,2,#,LSR,1,-,-,-,-,JMP,3,-,EOR,3,-,LSR,3,-
2070 DATA BVC,2,R,EOR,2,()Y,-,-,-,-,-,-,-,-,-,EOR,2,X,LSR,2,X,-,-,-
2080 DATA CLI,1,-,EOR,3,Y,-,-,-,-,-,-,-,-,-,EOR,3,X,LSR,3,X
2090 DATA RTS,1,-,ADC,2,(X),-,-,-,-,-,-,-,-,-,ADC,2,-,ROR,2,-,-,-,-
2100 DATA PLA,1,-,ADC,2,#,ROR,1,-,-,-,-,JMP,3,(),ADC,3,-,ROR,3,-
2110 DATA BVS,2,R,ADC,2,()Y,-,-,-,-,-,-,-,-,-,ADC,2,X,ROR,2,X,-,-,-
2120 DATA SEI,1,-,ADC,3,Y,-,-,-,-,-,-,-,-,-,ADC,3,X,ROR,3,X
2130 DATA -,-,-,STA,2,(X),-,-,-,-,-,-,STY,2,-,STA,2,-,STX,2,-,-,-,-
2140 DATA DEY,1,-,-,-,-,TXA,1,-,-,-,-,STY,3,-,STA,3,-,STX,3,-
2150 DATA BCC,2,R,STA,2,()Y,-,-,-,-,-,-,STY,2,X,STA,2,X,STX,2,Y,-,-,-
2160 DATA TYA,1,-,STA,3,Y,TXS,1,-,-,-,-,-,-,-,STA,3,X,-,-,-
2170 DATA LDY,2,#,LDA,2,(X),LDX,2,#,-,-,-,LDY,2,-,LDA,2,-,LDX,2,-,-,-,
2180 DATA TAY,1,-,LDA,2,#,TAX,1,-,-,-,-,LDY,3,-,LDA,3,-,LDX,3,-
2190 DATA BCS,2,R,LDA,2,()Y,-,-,-,-,-,-,LDY,2,X,LDA,2,X,LDX,2,Y,-,-,-
2200 DATA CLV,1,-,LDA,3,Y,TSX,1,-,-,-,-,LDY,3,X,LDA,3,X,LDX,3,Y
2210 DATA CPY,2,#,CMP,2,(X),-,-,-,-,-,-,CPY,2,-,CMP,2,-,DEC,2,-,-,-,-
2220 DATA INY,1,-,CMP,2,#,DEX,1,-,-,-,-,CPY,3,-,CMP,3,-,DEC,3,-
2230 DATA BNE,2,R,CMP,2,()Y,-,-,-,-,-,-,-,-,-,CMP,2,X,DEC,2,X,-,-,-
2240 DATA CLD,1,-,CMP,3,Y,-,-,-,-,-,-,-,-,-,CMP,3,X,DEC,3,X
2250 DATA CPX,2,#,SBC,2,(X),-,-,-,-,-,-,CPX,2,-,SBC,2,-,INC,2,-,-,-,-
2260 DATA INX,1,-,SBC,2,#,NOP,1,-,-,-,-,CPX,3,-,SBC,3,-,INC,3,-
2270 DATA BEQ,2,R,SBC,2,()Y,-,-,-,-,-,-,-,-,-,SBC,2,X,INC,2,X,-,-,-
2280 DATA SED,1,-,SBC,3,Y,-,-,-,-,-,-,-,-,-,SBC,3,X,INC,3,X
2290 REM ..................................... HEXCODE-DATEN
2300 DATA A2,00,8E,01,C8,8E,02,C8,18,AD,00,C8,6D,00,C8,8D,00,C8
2310 DATA 18,AD,01,C8,7D,03,C8,8D,01,C8,E8,AD,02,C8
2320 DATA 7D,03,C8,8D,02,C8,E8,EC,00,C8,D0,E6,00
2330 DATA $: REM ..... ENDEMARKIERUNG
```

BASIC-Ladeprogramm-Generator

Mit diesem Programm können Sie aus einem im Speicher stehenden Maschinenprogramm ein BASIC-Ladeprogramm erzeugen, das das Maschinenprogramm in DATA-Zeilen enthält.

Dazu geben Sie nach dem Programmstart lediglich den Namen sowie die Anfangs- und Endadresse des betreffenden Programms (dezimal) an. Sie haben außerdem die Möglichkeit, das Programmlisting auf dem Bildschirm oder dem Drucker ausgeben zu lassen.

```
10 REM ............................. BASIC-LADEPROGRAMM-GENERATOR
20 PRINT"■":PRINT TAB(5)"■ BASIC-LADEPROGRAMM-GENERATOR ":PRINT
30 PRINT:PRINT:INPUT"PROGRAMM-NAME";N$
40 PRINT:INUT"ANFANGSADRESSE, ENDADRESSE";A1,A2
50 PRINT:INPUT"B)ILDSCHIRM ODER D)RUCKER-AUSGABE";F$
60 IF F$="D" THEN OPEN 5,4,5: CMD 5: GOTO 80
70 PRINT"■"
80 PRINT" 1 REM .......... ";N$;
90 N=A2-A1
100 FOR I=0 TO N
110 IF I/10<>INT(I/10) THEN PRINT",";:GOTO 130
120 Z=Z+10:PRINT:PRINT Z;"DATA ";
130 GOSUB 250
140 NEXT I
150 Z=Z+10:PRINT:PRINT Z;"FOR I =";A1;"TO";A2
160 Z=Z+10:PRINT Z;"READ K: POKE I,K"
170 Z=Z+10:PRINT Z;"SUM=SUM+K"
180 Z=Z+10:PRINT Z;"NEXT I"
190 Z=Z+10:PRINT Z;
200 PRINT"IF SUM <>";SUM;"THEN PRINT ";CHR$(34);"PRUEFSUMME ";
210 PRINT"FALSCH!";CHR$(34);": END
220 Z=Z+10:PRINT Z;"PRINT ";CHR$(34);"PRUEFSUMME RICHTIG!";CHR$(34)
230 IF F$="D" THEN PRINT#5: CLOSE 5
240 END
250 A=PEEK(I+A1): SUM=SUM+A
260 A$=STR$(A)
270 IF LEFT$(A$,1)=" " THEN A$=RIGHT$(A$,LEN(A$)-1)
280 IF RIGHT$(A$,1)=" " THEN A$=LEFT$(A$,LEN(A$)-1)
290 IF LEN(A$)<3 THEN A$=" "+A$:GOTO 290
300 PRINT A$;
310 RETURN

READY.
```

```
                        ;*****************************
                        ;*                           *
                        ;* 16-BIT ROUTINEN FUER C-64 *
                        ;*                           *
                        ;*****************************

                        ;FILENAME... ASSI
                        ;LAST UPDATE 5-OCT-84

                        ;TRENG

                        ;EQUATES

14:   C000              GETIN   =   $FFE4
15:   C000              CHRIN   =   $FFCF
16:   C000              BITMAP  =   $D011
17:   C000              SCRMEM  =   $D018
18:   C000              PAG1    =   $100
19:   C000              GREG    =   $A3
20:   C000              R0      =   $4B
21:   C000              R1      =   R0+2
22:   C000              R2      =   R1+2
23:   C000              R3      =   R2+2
24:   C000              R4      =   R3+2
25:   C000              R5      =   R4+2
26:   C000              R6      =   R5+2
27:   C000              R7      =   R6+2
28:   C000              RSAV    =   R7+2
29:   C000              FNC     =   $FB
30:   C000              REGADR  =   $FC
31:   C000              PAR     =   $FD
```

```
                        ;INTERPRETER FUER 16-BIT-ROUTINEN

36:   C000 20 88 C0 ASSI    JSR  GETADR  ;RUECKSPRUNGADRESSE INS GREG
37:   C003 A0 00            LDY  #$00    ;FUER INDIREKTES LADEN VORBEREITEN
38:   C005 B1 A3            LDA  (GREG),Y  ;FUNKTIONSNUMMER HOLEN
39:   C007 29 20            AND  #$20    ;REM 4-BYTE-FUNKTION?
40:   C009 F0 19            BEQ  ASSI1   ;NEIN, NUR 2-BYTE-FUNKTION

                        ;4-BYTE-FUNKTION
43:   C00B 20 97 C0         JSR  INCG    ;RUECKSPRUNGADRESSE KORRIGIEREN
44:   C00E 20 97 C0         JSR  INCG    ;PARAMETERADRESSE -3
45:   C011 20 97 C0         JSR  INCG
46:   C014 20 A7 C0         JSR  PUTADR  ;KORRIGIERTE RUECKSPRUNGADRESSE ZURUECKSCHREIBEN
47:   C017 20 9E C0         JSR  DECG    ;PARAMETERADRESSE WIEDERHERSTELLEN
48:   C01A 20 9E C0         JSR  DECG
49:   C01D 20 9E C0         JSR  DECG
50:   C020 A0 03            LDY  #$03    ;4 BYTES UMSPEICHERN (ZAEHLER)
51:   C022 D0 0B            BNE  GETPAR  ;UEBERGABE-PARAMETER UMSPEICHERN

                        ;2-BYTE-FUNKTION
54:   C024 20 97 C0 ASSI1   JSR  INCG    ;RUECKSPRUNGADRESSE KORRIGIEREN
55:   C027 20 A7 C0         JSR  PUTADR  ;RUECKSPRUNGADRESSE ZURUECKSPEICHERN
56:   C02A 20 9E C0         JSR  DECG    ;PARAMETERADRESSE WIEDERHERSTELLEN
57:   C02D A0 01            LDY  #$01    ;2 BYTES UMSPEICHERN (ZAEHLER)

                        ;PARAMETER UMSPEICHERN
59:   C02F
                        ;FUNKTIONSNUMMER IN FNC
                        ;REGISTERINDEX IN   REGADR (HIGH-NIBBLE FUER QUELLE)
                        ;PARAMETER IN       PAR    (2 BYTES)
63:   C02F B1 A3    GETPAR  LDA  (GREG),Y
64:   C031 99 FB 00         STA  FNC,Y
65:   C034 88               DEY
66:   C035 10 F8            BPL  GETPAR

                        ;SPRUNG ZUR GEWAEHLTEN FUNKTION DURCH JSR
                        ;ZUERST DIE FUNKTIONS-ANFANGSADRESSE -1 IN DEN STACK SCHIEBEN,
                        ;DANN JUMP TO SUBROUTINE

72:   C037 0A               ASL          ;#2 = OFFSET IN TABELLE
73:   C038 A8               TAY          ;OFFSET FUER INDIZIERTES LADEN
74:   C039 B9 41 C0         LDA  FNCTAB-1,Y  ;LOW-BYTE ...
75:   C03C 48               PHA          ;IN DEN STACK
76:   C03D B9 40 C0         LDA  FNCTAB-2,Y  ;HIGH-BYTE ...
77:   C040 48               PHA          ;AUCH
78:   C041 60         DOIT   RTS          ;VERZWEIGUNG ZUR GEWAEHLTEN FUNKTION
```

```
                        ;ADRESS-TABELLE DER 16-BIT-ROUTINEN

83:    C042 E6 C0   FNCTAB   .WORDCLAREG-1   ;1 CLEAR ALL REGISTERS
84:    C044 F0 C0            .WORDCLREG-1    ;2 CLEAR REGISTER
85:    C046 FA C0            .WORDINCREG-1   ;3 INCREMENT REGISTER
86:    C048 04 C1            .WORDDECREG-1   ;4 DECREMENT REGISTER
87:    C04A 14 C1            .WORDSRREG-1    ;5 SHIFT RIGHT REGISTER
88:    C04C 22 C1            .WORDSLREG-1    ;6 SHIFT LEFT REGISTER
89:    C04E 30 C1            .WORDCSNREG-1   ;7 CHANGE SIGN OF REGISTER
90:    C050 40 C0            .WORDDOIT-1     ;8
91:    C052 40 C0            .WORDDOIT-1     ;9
92:    C054 40 C0            .WORDDOIT-1     ;10
93:    C056 41 C1            .WORDTRAREG-1   ;11 TRANSFER REGISTER
94:    C058 50 C1            .WORDCMPREG-1   ;12 COMPARE REGISTER
95:    C05A 62 C1            .WORDADD-1      ;13 ADDITION
96:    C05C 76 C1            .WORDSUB-1      ;14 SUBTRACTION
97:    C05E 40 C0            .WORDDOIT-1     ;15
98:    C060 40 C0            .WORDDOIT-1     ;16
99:    C062 40 C0            .WORDDOIT-1     ;17
100:   C064 40 C0            .WORDDOIT-1     ;18
101:   C066 40 C0            .WORDDOIT-1     ;19
102:   C068 40 C0            .WORDDOIT-1     ;20
103:   C06A 40 C0            .WORDDOIT-1     ;21
104:   C06C 40 C0            .WORDDOIT-1     ;22
105:   C06E 40 C0            .WORDDOIT-1     ;23
106:   C070 40 C0            .WORDDOIT-1     ;24
107:   C072 40 C0            .WORDDOIT-1     ;25
108:   C074 40 C0            .WORDDOIT-1     ;26
109:   C076 40 C0            .WORDDOIT-1     ;27
110:   C078 40 C0            .WORDDOIT-1     ;28
111:   C07A 40 C0            .WORDDOIT-1     ;29
112:   C07C 40 C0            .WORDDOIT-1     ;30
113:   C07E 40 C0            .WORDDOIT-1     ;31
114:   C080 8D C1            .WORDSETREG-1   ;32 SET REGISTER
115:   C082 99 C1            .WORDLRIND-1    ;33 LOAD REGISTER INDIRECT
116:   C084 A7 C1            .WORDSTOREG-1   ;34 STORE REGISTER INDIRECT
117:   C086 40 C0            .WORDDOIT-1     ;35
```

```
                        ;ALLGEMEIN VERWENDETE ROUTINEN FUER DEN 16-BIT-INTERPRETER


                        ;RUECKSPRUNGADRESSE HOLEN (GET ADDRESS)
                        ;RETURN MIT RUECKSPRUNGADRESSE IN GREG
125:  C088 BA       GETADR  TSX           ;AKT. STACKPOINTER
126:  C089 E8               INX
127:  C08A E8               INX
128:  C08B E8               INX           ;POINTER FUER RUECKSPRUNGADRESSE -1
129:  C08C BD 00 01         LDA  PAG1,X   ;RUECKSPRUNGADRESSE -1 ...
130:  C08F 85 A3            STA  GREG
131:  C091 E8               INX
132:  C092 BD 00 01         LDA  PAG1,X
133:  C095 85 A4            STA  GREG+1   ;INS GENERALREGISTER

                        ;INCREMENT GENERALREGISTER
136:  C097 E6 A3    INCG    INC  GREG     ;ZUERST DAS LOW-BYTE
137:  C099 D0 02            BNE  INCGX    ;LOW-BYTE <> 0, FERTIG!
138:  C09B E6 A4            INC  GREG+1   ;HIGH-BYTE AUCH INKREMENTIEREN
139:  C09D 60       INCGX   RTS

                        ;DECREMENT GENERALREGISTER
142:  C09E A5 A3    DECG    LDA  GREG     ;ZUERST DAS LOW-BYTE
143:  C0A0 D0 02            BNE  DECG1    ;LOW-BYTE <> 0, NUR LOW-BYTE DEKREMENTIEREN
144:  C0A2 C6 A4            DEC  GREG+1   ;HIGH-BYTE AUCH DEKREMENTIEREN
145:  C0A4 C6 A3    DECG1   DEC  GREG     ;LOW-BYTE DEKREMENTIEREN
146:  C0A6 60               RTS

                        ;RUECKSPRUNGADRESSE IN DEN STACK ZURUECKSCHREIBEN
                        ;RUECKSPRUNGADRESSE -1 IN GREG
150:  C0A7 BA       PUTADR  TSX           ;AKT. STACKPOINTER
151:  C0A8 E8               INX
152:  C0A9 E8               INX
153:  C0AA E8               INX           ;OFFSET DER RUECKSPRUNGADRESSE IM STACK
154:  C0AB A5 A3            LDA  GREG     ;LOW-BYTE DER RUECKSPRUNGADRESSE ...
155:  C0AD 9D 00 01         STA  PAG1,X   ;IN DEN STACK
156:  C0B0 E8               INX
157:  C0B1 A5 A4            LDA  GREG+1   ;HIGH-BYTE
158:  C0B3 9D 00 01         STA  PAG1,X
159:  C0B6 60               RTS
```

```
                    ;LADE GENERALREGISTER INDIREKT
                    ;AUFRUF MIT OPERANDENADRESSE IN GREG
                    ;RETURN MIT OPERAND IN GREG
164:  C0B7 A0 00  LGIND   LDY  #$00    ;INDEX=0
165:  C0B9 B1 A3          LDA  (GREG),Y  ;LOW-BYTE DES OPERANDEN
166:  C0BB 48             PHA
167:  C0BC C8             INY          ;INDEX=1
168:  C0BD B1 A3          LDA  (GREG),Y  ;HIGH-BYTE DES OPERANDEN
169:  C0BF 85 A4          STA  GREG+1
170:  C0C1 68             PLA
171:  C0C2 85 A3          STA  GREG
172:  C0C4 60             RTS

                    ;INDEX DES QUELLENREGISTERS (SOURCE REGISTER) HOLEN
                    ;RETURN MIT REGISTERNUMMER*2 IM X-INDEXREGISTER
                    ;GET INDEX FOR SRC.-REG.
177:  C0C5 A5 FC  GETSR   LDA  REGADR
178:  C0C7 29 70          AND  #$70    ;REGISTERNUMMER DES ZIELREGISTERS AUSBLENDEN
179:  C0C9 4A             LSR
180:  C0CA 4A             LSR
181:  C0CB 4A             LSR          ;3X NACH RECHTS SCHIEBEN = REGISTERNUMMER*2
182:  C0CC AA             TAX
183:  C0CD 60             RTS

                    ;INDEX DES ZIELREGISTERS (DESTINATION REGISTER) HOLEN
                    ;RETURN MIT REGISTERNUMMER*2 IM X-INDEXREGISTER
                    ;GET INDEX FOR DST.-REG.
188:  C0CE A5 FC  GETDI   LDA  REGADR
189:  C0D0 29 07          AND  #$07    ;REGISTERNUMMER DES QUELLENREGISTERS AUSBLENDEN
190:  C0D2 0A             ASL          ;1X NACH LINKS SCHIEBEN = REGISTERNUMMER*2
191:  C0D3 AA             TAX
192:  C0D4 60             RTS

                    ;LADE GENERALREGISTER MIT SOURCEREGISTER
                    ;AUFRUF MIT INDEX DES 16-BIT-REGISTERS IN REGADR
                    ;  ODER MIT INDEX DES 16-BIT-REGISTERS IM X-INDEXREGISTER,
                    ;       WOBEI LOADR DIE EINSPRUNGSTELLE IST
                    ;LOAD GENERALREG. WITH SRC.-REG
199:  C0D5 20 C5 C0 LOADS   JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
200:  C0D8 B5 4B    LOADR   LDA  R0,X    ;LOW-BYTE
201:  C0DA 85 A3          STA  GREG
202:  C0DC B5 4C          LDA  R0+1,X  ;HIGH-BYTE
203:  C0DE 85 A4          STA  GREG+1
204:  C0E0 60             RTS

                    ;LADE GENERALREGISTER MIT ZIELREGISTER
                    ;AUFRUF MIT INDEX DES 16-BIT-REGISTERS IN REGADR
208:  C0E1 20 CE C0 LOADD   JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
209:  C0E4 4C D8 C0         JMP  LOADR
```

```
                        ;ALLE REGISTER LOESCHEN (CLEAR ALL REGISTERS)
212:  C0E7 A9 00    CLAREG  LDA  #$00
213:  C0E9 A2 10            LDX  #$10     ;16 BYTES
214:  C0EB 95 4B    CLR     STA  R0,X
215:  C0ED CA               DEX
216:  C0EE 10 FB            BPL  CLR
217:  C0F0 60               RTS

                        ;REGISTER LOESCHEN (CLEAR REGISTER)
220:  C0F1 20 C5 C0 CLREG   JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
221:  C0F4 A9 00            LDA  #$00
222:  C0F6 95 4B            STA  R0,X     ;LOW-BYTE
223:  C0F8 95 4C            STA  R0+1,X   ;HIGH-BYTE
224:  C0FA 60               RTS

                        ;INCREMENT REGISTER
227:  C0FB 20 C5 C0 INCREG  JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
228:  C0FE F6 4B            INC  R0,X
229:  C100 D0 02            BNE  INRX
230:  C102 F6 4C            INC  R0+1,X
231:  C104 60       INRX    RTS

                        ;DECREMENT REGISTER
234:  C105 20 C5 C0 DECREG  JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
235:  C108 B5 4B            LDA  R0,X
236:  C10A D0 02            BNE  DECR1
237:  C10C D6 4C            DEC  R0+1,X
238:  C10E D6 4B    DECR1   DEC  R0,X
239:  C110 B5 4B            LDA  R0,X     ;KONDITIONSBIT (ZERO FLAG) AKTUALISIEREN
240:  C112 15 4C            ORA  R0+1,X
241:  C114 60               RTS

                        ; SHIFT RIGHT REGISTER
244:  C115 20 C5 C0 SRREG   JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
245:  C118 B5 4C            LDA  R0+1,X
246:  C11A 4A               LSR          ;SHIFT RIGHT HIGH-BYTE, MSB IN CARRY
247:  C11B 95 4C            STA  R0+1,X  ;ZURUECKSCHREIBEN
248:  C11D B5 4B            LDA  R0,X    ;LOW-BYTE
249:  C11F 6A               ROR          ;ROTIEREN NACH RECHTS DURCH CARRY
250:  C120 95 4B            STA  R0,X    ;ZURUECKSCHREIBEN
251:  C122 60               RTS

                        ; SHIFT LEFT REGISTER
254:  C123 20 C5 C0 SLREG   JSR  GETSR    ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
255:  C126 B5 4B            LDA  R0,X    ;LOW-BYTE
256:  C128 0A               ASL          ;SHIFT LEFT LOW-BYTE, MSB IN CARRY
257:  C129 95 4B            STA  R0,X    ;ZURUECKSCHREIBEN
258:  C12B B5 4C            LDA  R0+1,X  ;HIGH-BYTE
259:  C12D 2A               ROL          ;ROTIEREN NACH LINKS DURCH CARRY
260:  C12E 95 4C            STA  R0+1,X  ;ZURUECKSCHREIBEN
261:  C130 60               RTS
```

```
                        ;REGISTER-VORZEICHEN WECHSELN (CHANGE SIGN OF REGISTER)
264:  C131 20 C5 C0 CSNREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
265:  C134 38               SEC          ;SUBTRAKTION VORBEREITEN
266:  C135 A9 00            LDA  #$00    ;NULL ...
267:  C137 F5 4B            SBC  R0,X    ;MINUS REGISTER
268:  C139 95 4B            STA  R0,X    ;ZURUECKSCHREIBEN
269:  C13B A9 00            LDA  #$00    ;DESGLEICHEN FUER HIGH-BYTE
270:  C13D F5 4C            SBC  R0+1,X
271:  C13F 95 4C            STA  R0+1,X
272:  C141 60               RTS

                        ;QUELLENREGISTER -> ZIELREGISTER (TRANSFER REGISTER)
                        ;INHALT DES QUELLENREGISTERS WIRD NICHT GEAENDERT
276:  C142 20 D5 C0 TRAREG  JSR  LOADS   ;GREG MIT INHALT DES QUELLENREGISTERS LADEN
277:  C145 20 CE C0 SAVG    JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
278:  C148 A5 A3    SAVREG  LDA  GREG    ;INHALT DES GENERALREGISTERS INS GEWUENSCHTE
279:  C14A 95 4B            STA  R0,X    ;REGISTER SPEICHERN
280:  C14C A5 A4            LDA  GREG+1
281:  C14E 95 4C            STA  R0+1,X
282:  C150 60               RTS

                        ;REGISTER VERGLEICHEN (COMPARE REGISTER)
                        ;DIE KONDITIONSBITS WERDEN WIE BEIM 8-BIT-BEFEHL AKTUALISIERT
286:  C151 20 C5 C0 CMPREG  JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM AKKU ...
287:  C154 A8               TAY          ;UND JETZT IM Y-INDEXREGISTER
288:  C155 20 CE C0         JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
289:  C158 B9 4B 00         LDA  R0,Y    ;JETZT VERGLEICHEN
290:  C15B D5 4B            CMP  R0,X
291:  C15D B9 4C 00         LDA  R0+1,Y
292:  C160 F5 4C            SBC  R0+1,X
293:  C162 60               RTS

                        ;ADDITION VON QUELLEN- UND ZIELREGISTER
                        ;ERGEBNIS IM QUELLENREGISTER, ZIELREG. BLEIBT UNVERAENDERT
297:  C163 20 E1 C0 ADD     JSR  LOADD   ;LADE GENERALREGISTER MIT ZIELREGISTER
298:  C166 20 C5 C0         JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
299:  C169 18               CLC          ;ADDITION VORBEREITEN
300:  C16A B5 4B            LDA  R0,X    ;LOW-BYTE DES QUELLENREGISTERS ZUM ...
301:  C16C 65 A3            ADC  GREG    ;LOW-BYTE DES GENERALREGISTERS ADDIEREN ...
302:  C16E 95 4B            STA  R0,X    ;UND INS QUELLENREGISTER SPEICHERN
303:  C170 B5 4C            LDA  R0+1,X  ;DAS GLEICHE AUCH FUER DAS HIGH-BYTE
304:  C172 65 A4            ADC  GREG+1
305:  C174 95 4C            STA  R0+1,X
306:  C176 60               RTS
```

```
                        ;ZIELREGISTER VON QUELLENREGISTER SUBTRAHIEREN
                        ;ERGEBNIS IM QUELLENREGISTER, ZIELREG. BLEIBT UNVERAENDERT
310:  C177 20 CE C0 SUB      JSR  GETDI   ;INDEX DES ZIELREGISTERS ...
311:  C17A A8                TAY          ;INS Y-INDEXREGISTER
312:  C17B 20 C5 C0          JSR  GETSR   ;INDEX DES QUELLENREGISTERS INS X-INDEXREGISTER
313:  C17E 38                SEC          ;SUBTRAKTION VORBEREITEN
314:  C17F B5 4B             LDA  R0,X
315:  C181 F9 4B 00          SBC  R0,Y
316:  C184 95 4B             STA  R0,X
317:  C186 B5 4C             LDA  R0+1,X
318:  C188 F9 4C 00          SBC  R0+1,Y
319:  C18B 95 4C             STA  R0+1,X
320:  C18D 60                RTS

                        ;REGISTER MIT PARAMETER VORBELEGEN
                        ;(SET REGISTER WITH PARAMETER)
324:  C18E 20 CE C0 SETREG   JSR  GETDI   ;INDEX DES ZIELREGISTERS IM X-INDEXREGISTER
325:  C191 A5 FD             LDA  PAR     ;LOW-BYTE DES PARAMETERS ...
326:  C193 95 4B             STA  R0,X    ;INS REGISTER SPEICHERN
327:  C195 A5 FE             LDA  PAR+1   ;DAS GLEICHE AUCH FUER ...
328:  C197 95 4C             STA  R0+1,X  ;HIGH-BYTE
329:  C199 60                RTS

                        ;REGISTER INDIREKT LADEN (LOAD REGISTER INDIRECT)
                        ;DIE PARAMETER-ADRESSE STEHT IN PAR
                        ;RETURN MIT OPERANDEN IN GREG UND IM GEWUENSCHTEN REGISTER
334:  C19A A5 FD    LRIND    LDA  PAR     ;OPERANDENADRESSE NACH ...
335:  C19C 85 A3             STA  GREG    ;GENERALREGISTER ...
336:  C19E A5 FE             LDA  PAR+1   ;SPEICHERN
337:  C1A0 85 A4             STA  GREG+1
338:  C1A2 20 B7 C0          JSR  LGIND   ;LADE GENERALREGISTER INDIREKT ...
339:  C1A5 4C 45 C1          JMP  SAVG    ;UND SPEICHERE INS GEWUENSCHTE REGISTER

                        ;REGISTER INDIREKT SPEICHERN (STORE REGISTER INDIRECT)
                        ;DIE SPEICHERADRESSE STEHT IN PAR
343:  C1A8 20 C5 C0 STOREG   JSR  GETSR   ;INDEX DES QUELLENREGISTERS IM X-INDEXREGISTER
344:  C1AB A0 00             LDY  #$00    ;VORBEREITUNG FUER INDIREKTES LADEN
345:  C1AD B5 4B             LDA  R0,X    ;LOW-BYTE DES REGISTERS HOLEN UND ...
346:  C1AF 91 FD             STA  (PAR),Y  ;INDIREKT SPEICHERN
347:  C1B1 C8                INY          ;NAECHSTES BYTE (HIGH-BYTE)
348:  C1B2 B5 4C             LDA  R0+1,X
349:  C1B4 91 FD             STA  (PAR),Y
350:  C1B6 60                RTS
```

```
                    ;FILL MEMORY
                    ;STARTADRESSE = R0
                    ;ENDADRESSE   = R1
                    ;DATA IN ACCUMULATOR
357:  C1B7 A0 00    FILL    LDY  #$00    ;INDIREKTES SPEICHERN VORBEREITEN
358:  C1B9 91 4B    FL1     STA  (R0),Y
359:  C1BB E6 4B            INC  R0
360:  C1BD D0 02            BNE  TSTEND
361:  C1BF E6 4C            INC  R0+1
362:  C1C1 A6 4C    TSTEND  LDX  R0+1
363:  C1C3 E4 4E            CPX  R1+1
364:  C1C5 D0 F2            BNE  FL1
365:  C1C7 A6 4B            LDX  R0
366:  C1C9 E4 4D            CPX  R1
367:  C1CB 90 EC            BCC  FL1
368:  C1CD 60               RTS
```

```
                    ;LINIE ZEICHNEN/LOESCHEN (PLOT LINE/ERASE LINE)

                    ;STARTKOORDINATE (XS,YS)  R0,R1
                    ;ENDKOORDINATE   (XE,YE)  R2,R3

                    ;RSAV WIRD ALS ADRESSPEICHER BENUTZT (BASIS)

378:  C1CE A9 00    PLOT     LDA  #$00
379:  C1D0 F0 03             BEQ  ERASE1
380:  C1D2 00        COLOR    .BYTE$00
381:  C1D3 A9 FF     ERASE    LDA  #$FF
382:  C1D5 8D D2 C1  ERASE1   STA  COLOR
                    ;DELTA X,Y BILDEN
384:  C1D8 20 93 C2           JSR  TOGGLE  ;1. PUNKT
385:  C1DB 20 00 C0           JSR  ASSI
386:  C1DE 0E 20              .BYTE$0E,$20  ;DELTAX -> R2
387:  C1E0 20 00 C0           JSR  ASSI
388:  C1E3 0E 31              .BYTE$0E,$31  ;DELTAY -> R3
                    ;INKREMENT FUER X BESTIMMEN
390:  C1E5 A5 50    GIFX     LDA  R2+1    ;DELTAX POS ODER NEG ?
391:  C1E7 10 0E             BPL  XPOS    ;DELTAX IST POSITIV
392:  C1E9 A9 FF             LDA  #$FF
393:  C1EB 85 53             STA  R4
394:  C1ED 85 54             STA  R4+1    ;INKREMENT FUER X IST -1
395:  C1EF 20 00 C0           JSR  ASSI
396:  C1F2 07 20              .BYTE$07,$20  ;VORZEICHEN-WECHSEL
397:  C1F4 4C FE C1           JMP  GIFY
398:  C1F7 20 00 C0 XPOS      JSR  ASSI
399:  C1FA 20 04 01           .BYTE$20,$04,$01,$00  ;INKREMENT FUER X IST +1
                    ;INKREMENT FUER Y BESTIMMEN
401:  C1FE A5 52    GIFY     LDA  R3+1    ; DELTAY POS ODER NEG ?
402:  C200 10 0E             BPL  YPOS    ;DELTAY IST POSITIV
403:  C202 A9 FF             LDA  #$FF
404:  C204 85 55             STA  R5
405:  C206 85 56             STA  R5+1    ;INKREMENT FUER Y IST -1
406:  C208 20 00 C0           JSR  ASSI
407:  C20B 07 30              .BYTE$07,$30  ;VORZEICHEN-WECHSEL
408:  C20D 4C 17 C2           JMP  DRAW
409:  C210 20 00 C0 YPOS      JSR  ASSI
410:  C213 20 05 01           .BYTE$20,$05,$01,$00  ;INKREMENT FUER Y IST +1
```

```
412:   C217 20 00 C0 DRAW     JSR  ASSI     ; DELTAX > DELTAY ?
413:   C21A 0C 23             .BYTE$0C,$23  ;VERGLEICHE R2 MIT R3
414:   C21C B0 03             BCS  DRAWX    ; DELTAX IST GROESSER, ZEICHNEN NACH X-ACHSE
415:   C21E 4C 5E C2          JMP  DRAWY

417:   C221 A5 4F    DRAWX    LDA  R2       ;IST DELTAX = 0 ?
418:   C223 D0 04             BNE  DRX1
419:   C225 A5 50             LDA  R2+1
420:   C227 F0 69             BEQ  DRFINI   ;WENN JA, FERTIG

422:   C229 20 00 C0 DRX1     JSR  ASSI     ;KORREKTUR-REGISTER IST R6
423:   C22C 0B 26             .BYTE$0B,$26  ;DELTAX -> R6
424:   C22E 20 00 C0          JSR  ASSI
425:   C231 05 60             .BYTE$05,$60  ;R6/2 -> KORREKTUR
426:   C233 20 00 C0          JSR  ASSI
427:   C236 0B 27             .BYTE$0B,$27  ;R7 = ZAEHLER

429:   C238 20 00 C0 DRX2     JSR  ASSI
430:   C23B 0D 63             .BYTE$0D,$63  ;KORR+DY->KORR
431:   C23D 20 00 C0          JSR  ASSI
432:   C240 0C 62             .BYTE$0C,$62  ;VERGLEICHE KORR,DX
433:   C242 90 0A             BCC  DRX3     ;KLEINER, DY BLEIBT
                        ; KORR>DX, BEIDE KOORDINATEN ANDERN SICH
435:   C244 20 00 C0          JSR  ASSI
436:   C247 0E 62             .BYTE$0E,$62 ;KORR-DX->KORR
437:   C249 20 00 C0          JSR  ASSI
438:   C24C 0D 15             .BYTE$0D,$15  ;AENDERT Y-KOORD.
439:   C24E 20 00 C0 DRX3     JSR  ASSI
440:   C251 0D 04             .BYTE$0D,$04  ;AENDERT X-KOORD.
441:   C253 20 93 C2          JSR  TOGGLE   ;SET OR CLEAR BIT
442:   C256 20 00 C0          JSR  ASSI
443:   C259 04 70             .BYTE$04,$70  ;WEG DEKREMENTIEREN
444:   C25B D0 DB             BNE  DRX2
445:   C25D 60                RTS

445:   C25D 60                RTS
```

```
447:  C25E 20 00 C0 DRAWY    JSR  ASSI     ;KORREKTUR-REGISTER IST R6
448:  C261 0B 36             .BYTE$0B,$36  ;DY -> R6
449:  C263 20 00 C0          JSR  ASSI
450:  C266 05 60             .BYTE$05,$60  ;R6/2 -> KORREKTUR
451:  C268 20 00 C0          JSR  ASSI
452:  C26B 0B 37             .BYTE$0B,$37  ;R7 = ZAEHLER

454:  C26D 20 00 C0 DRY2     JSR  ASSI
455:  C270 0D 62             .BYTE$0D,$62  ;KORR+DX -> KORR
456:  C272 20 00 C0          JSR  ASSI
457:  C275 0C 63             .BYTE$0C,$63  ;VERGLEICHE KORR,DY
458:  C277 90 0A             BCC  DRY3     ;KLEINER, X-KOORD. BLEIBT
                       ; KORR>DY, BEIDE KOORD. ANDERN SICH
460:  C279 20 00 C0          JSR  ASSI
461:  C27C 0E 63             .BYTE$0E,$63  ;KORR-DY -> KORR
462:  C27E 20 00 C0          JSR  ASSI
463:  C281 0D 04             .BYTE$0D,$04  ;AENDERT X-KOORD.
464:  C283 20 00 C0 DRY3     JSR  ASSI
465:  C286 0D 15             .BYTE$0D,$15  ;AENDERT Y-KOORD.
466:  C288 20 93 C2          JSR  TOGGLE   ;SET OR CLEAR BIT
467:  C28B 20 00 C0          JSR  ASSI
468:  C28E 04 70             .BYTE$04,$70  ;WEG DEKREMENTIEREN
469:  C290 D0 DB             BNE  DRY2
470:  C292 60        DRFINI  RTS



474:  C293 2C D2 C1 TOGGLE   BIT  COLOR
475:  C296 30 0A             BMI  CLRBIT

                      ;SET GRAPHIC DOT
478:  C298 20 AE C2 SETBIT   JSR  GRAFAD
479:  C29B A0 00    DOBIT    LDY  #$00
480:  C29D 11 5B             ORA  (RSAV),Y
481:  C29F 91 5B             STA  (RSAV),Y
482:  C2A1 60                RTS

                      ;CLEAR GRAPHIC DOT
485:  C2A2 20 AE C2 CLRBIT   JSR  GRAFAD
486:  C2A5 49 FF             EOR  #$FF
487:  C2A7 A0 00             LDY  #$00
488:  C2A9 31 5B             AND  (RSAV),Y
489:  C2AB 91 5B             STA  (RSAV),Y
490:  C2AD 60                RTS
```

```
                    ;ADRESSE EINES GRAPHIKSPEICHERS BERECHNEN
                    ; X-KOORD. IN R0
                    ; Y-KOORD. IN R1
496:  C2AE A5 4D    GRAFAD  LDA  R1       ;Y-KOORD., 1 BYTE
497:  C2B0 4A               LSR
498:  C2B1 4A               LSR
499:  C2B2 4A               LSR           ;ZEILENNUMMER
500:  C2B3 A8               TAY
501:  C2B4 B9 E7 C2         LDA  HIGTAB,Y
502:  C2B7 18               CLC
503:  C2B8 65 4C            ADC  R0+1     ;X-KOORD. (HIGH-BYTE) ALS KORREKTUR
504:  C2BA 85 5C            STA  RSAV+1   ;SICHERN IN RSAV (HIGH-BYTE)
505:  C2BC 98               TYA
506:  C2BD 29 03            AND  #$03     ;ZEILENNUMMER MOD 4
507:  C2BF A8               TAY
508:  C2C0 A5 4B            LDA  R0
509:  C2C2 29 F8            AND  #$F8     ;INT(X-KOORD./8)*8
510:  C2C4 79 00 C3         ADC  LOWTAB,Y
511:  C2C7 85 5B            STA  RSAV
512:  C2C9 90 03            BCC  GRAF1
513:  C2CB E6 5C            INC  RSAV+1
514:  C2CD 18               CLC
515:  C2CE A5 4D    GRAF1   LDA  R1       ;Y-KOORD.
516:  C2D0 29 07            AND  #$07
517:  C2D2 65 5B            ADC  RSAV
518:  C2D4 85 5B            STA  RSAV

520:  C2D6 A5 4B    GETBIT  LDA  R0
521:  C2D8 29 07            AND  #$07
522:  C2DA A8               TAY
523:  C2DB A9 80            LDA  #$80
524:  C2DD C0 00            CPY  #$00
525:  C2DF F0 05            BEQ  GBOUT
526:  C2E1 18       GBLOOP  CLC
527:  C2E2 6A               ROR
528:  C2E3 88               DEY
529:  C2E4 D0 FB            BNE  GBLOOP
530:  C2E6 60       GBOUT   RTS

532:  C2E7 20 21 22 HIGTAB  .BYTE$20,$21,$22,$23,$25
533:  C2EC 26 27 28         .BYTE$26,$27,$28,$2A,$2B
534:  C2F1 2C 2D 2F         .BYTE$2C,$2D,$2F,$30,$31
535:  C2F6 32 34 35         .BYTE$32,$34,$35,$36,$37
536:  C2FB 39 3A 3B         .BYTE$39,$3A,$3B,$3C,$3E
537:  C300 00 40 80 LOWTAB  .BYTE$00,$40,$80,$C0
```

```
                        ;GRAFIK EINSCHALTEN
540:   C304 AD 11 D0 GON      LDA  BITMAP
541:   C307 8D 28 C3          STA  VIDEO1
542:   C30A A9 3B             LDA  #$3B
543:   C30C 8D 11 D0          STA  BITMAP
544:   C30F AD 18 D0          LDA  SCRMEM
545:   C312 8D 29 C3          STA  VIDEO2
546:   C315 A9 18             LDA  #$18
547:   C317 8D 18 D0          STA  SCRMEM
548:   C31A 60                RTS

                        ;GRAFIK AUSSCHALTEN
551:   C31B AD 28 C3 GOFF     LDA  VIDEO1
552:   C31E 8D 11 D0          STA  BITMAP
553:   C321 AD 29 C3          LDA  VIDEO2
554:   C324 8D 18 D0          STA  SCRMEM
555:   C327 60                RTS

557:   C328 00          VIDEO1   .BYTE$00
558:   C329 00          VIDEO2   .BYTE$00

                        ;GRAFIKSPEICHER LOESCHEN ($2000-$3FFF)
561:   C32A 20 00 C0 GCLR     JSR  ASSI
562:   C32D 20 00 00          .BYTE$20,$00,$00,$20  ;R0=$2000
563:   C331 20 00 C0          JSR  ASSI
564:   C334 20 01 FF          .BYTE$20,$01,$FF,$3F  ;R1=$3FFF
565:   C338 A9 00             LDA  #$00
566:   C33A 4C B7 C1          JMP  FILL

                        ;HINTERGRUNDFARBE SETZEN ($400-$7E7)
                        ;FARBE IM AKKU
570:   C33D 48          GCOL     PHA
571:   C33E 20 00 C0          JSR  ASSI
572:   C341 20 00 00          .BYTE$20,$00,$00,$04  ;R0=$400
573:   C345 20 00 C0          JSR  ASSI
574:   C348 20 01 E7          .BYTE$20,$01,$E7,$07  ;R1=$7E7
575:   C34C 68                PLA
576:   C34D 4C B7 C1          JMP  FILL
```

```
                        ;SPRITE-ROUTINEN

581:  C350              SPPMEM  =   $7F8    ;SPRITE POINTER MEMORY
582:  C350              SPPOSR  =   $D000   ;SPRITE POSITIONSREGISTER
583:  C350              SPCOLR  =   $D027   ;SPRITE COLOR REGISTER
584:  C350              SPENA   =   $D015   ;SPRITE ENABLE REGISTER
585:  C350              SPCHR   =   R0      ;SPRITE TABELLEN POINTER
586:  C350              SPMEM   =   R1      ;SPRITE MEMORY POINTER
587:  C350              LENGTH  =   R2      ;SPRITE ZEILENANZAHL
588:  C350              BITCNT  =   R3
589:  C350              SHIFTR  =   R3+1

591:  C350 00           SPCOL   .BYTE$00
592:  C351 00           SPNUM   .BYTE$00

                        ;SPRITE ON, SPRITE# IN (SPNUM)
595:  C352 AD 51 C3 SPRON   LDA  SPNUM   ;SPRITENUMMER HOLEN
596:  C355 20 77 C3         JSR  SETPOS  ;BIT-POSITION ENTSPR. SPRITENR. IN DEN AKKU
597:  C358 0D 15 D0         ORA  SPENA   ;SPRITE ...
598:  C35B 8D 15 D0         STA  SPENA   ;EINSCHALTEN

                        ;SPRITEFARBE BESTIMMEN (SET SPRITE COLOR)
601:  C35E AC 51 C3 SSPCOL  LDY  SPNUM   ;SPRITEFARBE IN DEN ...
602:  C361 AD 50 C3         LDA  SPCOL   ;DEM SPRITE ZUGEORDNETEN ...
603:  C364 99 27 D0         STA  SPCOLR,Y  ;FARBSPEICHER BRINGEN
604:  C367 60               RTS

                        ;SPRITE AUSSCHALTEN (SPRITE OFF), SPRITE-NR. IN (SPNUM)
607:  C368 AD 51 C3 SPOFF   LDA  SPNUM   ;SPRITENUMMER HOLEN
608:  C36B 20 77 C3         JSR  SETPOS  ;BIT-POSITION ENTSPR. SPRITENR. IN DEN AKKU
609:  C36E 49 FF            EOR  #$FF    ;SPRITE ...
610:  C370 2D 15 D0         AND  SPENA   ;AUS-
611:  C373 8D 15 D0         STA  SPENA   ;SCHALTEN
612:  C376 60               RTS

                        ;SET BIT IN ACCU
                        ;RETURN MIT BITPOSITION (=2↑SPRITENUMMER) IM AKKU
616:  C377 A8           SETPOS  TAY
617:  C378 A9 01            LDA  #$01
618:  C37A C8               INY
619:  C37B 88           SETP1   DEY          ;SPRITE-NR. HERUNTERZAEHLEN
620:  C37C F0 03            BEQ  SETPX   ;SOLANGE SPRITE-NR. <> 0 ...
621:  C37E 0A               ASL          ;AKKUINHALT (BITPOSITION) VERDOPPELN
622:  C37F D0 FA            BNE  SETP1
623:  C381 60           SETPX   RTS

625:  C382 03 03 03 SPHTAB  .BYTE$03,$03,$03
626:  C385 40 80 C0 SPLTAB  .BYTE$40,$80,$C0
627:  C388 0D 0E 0F SPBLCK  .BYTE$0D,$0E,$0F
```

```
                        ;GET SPRITE POINTER
630:   C38B AC 51 C3 SPPTR    LDY  SPNUM
631:   C38E B9 88 C3          LDA  SPBLCK,Y  ;POINTER AUF SPRITE-DEFINITIONSBLOCK ...
632:   C391 99 F8 07          STA  SPPMEM,Y  ;INS SPRITE POINTER MEMORY
633:   C394 B9 85 C3          LDA  SPLTAB,Y  ;ANFANGSADRESSE DER SPRITE-DEFINITIONSTABELLE ...
634:   C397 85 4D             STA  SPMEM   ;INS SPRITE MEMORY POINTER REGISTER
635:   C399 B9 82 C3          LDA  SPHTAB,Y
636:   C39C 85 4E             STA  SPMEM+1
637:   C39E 60                RTS

                        ;SPRITE ERSTELLEN (MAKE SPRITE)
640:   C39F 20 88 C0 SPMAKE   JSR  GETADR  ;ADRESSE DER ANFANGSADRESSE DER SPRITE-TABELLE HOLEN
641:   C3A2 20 97 C0          JSR  INCG
642:   C3A5 20 A7 C0          JSR  PUTADR  ;KORR. RUECKSPRUNGADRESSE ZURUECKSCHREIBEN
643:   C3A8 20 9E C0          JSR  DECG
644:   C3AB 20 B7 C0          JSR  LGIND   ;TABELLEN-ANFANGSADRESSE HOLEN UND ...
645:   C3AE A2 00             LDX  #$00    ;NACH R0 ...
646:   C3B0 20 48 C1          JSR  SAVREG  ;SPEICHERN
647:   C3B3 20 8B C3          JSR  SPPTR   ;SPRITE POINTER HOLEN
648:   C3B6 A0 00             LDY  #$00
649:   C3B8 A9 15             LDA  #$15    ;21 SPRITEZEILEN
650:   C3BA 85 4F             STA  LENGTH

652:   C3BC A2 18    SPM0     LDX  #$18    ;24 SPRITESPALTEN
653:   C3BE A9 08    SPM1     LDA  #$08
654:   C3C0 85 51             STA  BITCNT
655:   C3C2 84 52             STY  SHIFTR  ;SHIFT REGISTER LOESCHEN
656:   C3C4 B1 4B    SPM2     LDA  (SPCHR),Y  ;SPRITEZEICHEN AUS TABELLE HOLEN
657:   C3C6 E6 4B             INC  SPCHR   ;AUF NAECHSTES ZEICHEN ZEIGEN
658:   C3C8 D0 02             BNE  SPM22
659:   C3CA E6 4C             INC  SPCHR+1
660:   C3CC 29 7F    SPM22    AND  #$7F    ;MSB LOESCHEN
661:   C3CE C9 20             CMP  #$20    ;LEERZEICHEN STEHT FUER "SPRITE-PUNKT AUS"
662:   C3D0 F0 03             BEQ  SPM3
663:   C3D2 38                SEC
664:   C3D3 B0 01             BCS  SPM33
665:   C3D5 18       SPM3     CLC
666:   C3D6 26 52    SPM33    ROL  SHIFTR
667:   C3D8 CA       SPM4     DEX          ;NAECHSTER SPRITEPUNKT
668:   C3D9 F0 11             BEQ  SPM5    ;WENN JA, NAECHSTE SPRITEZEILE
669:   C3DB C6 51             DEC  BITCNT  ;NAECHSTES BIT
670:   C3DD D0 E5             BNE  SPM2    ;NAECHSTEN PUNKT IM SELBEN BYTE BEARBEITEN
671:   C3DF A5 52             LDA  SHIFTR  ;EIN BYTE FERTIG
672:   C3E1 91 4D             STA  (SPMEM),Y
673:   C3E3 E6 4D             INC  SPMEM
674:   C3E5 D0 02             BNE  SPM44
675:   C3E7 E6 4E             INC  SPMEM+1
676:   C3E9 4C BE C3 SPM44    JMP  SPM1
```

```
677:  C3EC C6 51    SPM5     DEC  BITCNT
678:  C3EE A5 52              LDA  SHIFTR
679:  C3F0 91 4D              STA  (SPMEM),Y
680:  C3F2 E6 4D              INC  SPMEM
681:  C3F4 D0 02              BNE  SPM55
682:  C3F6 E6 4E              INC  SPMEM+1
683:  C3F8 C6 4F    SPM55    DEC  LENGTH
684:  C3FA D0 C0              BNE  SPM0
685:  C3FC 60                 RTS

                        ;SPRITE POSITIONIERUNG
688:  C3FD          XKOR     =    R4
689:  C3FD          YKOR     =    R5
690:  C3FD AD 51 C3 SPPOS    LDA  SPNUM
691:  C400 0A                 ASL
692:  C401 A8                 TAY
693:  C402 A5 55              LDA  YKOR
694:  C404 99 01 D0           STA  SPPOSR+1,Y
695:  C407 A5 53              LDA  XKOR
696:  C409 99 00 D0           STA  SPPOSR,Y
697:  C40C A5 54              LDA  XKOR+1
698:  C40E 8D 10 D0           STA  SPPOSR+16
699:  C411 60                 RTS


                        ;SPRITE-DEFINITIONSTABELLE

704:  C412 2A 2A 2A QUADER   .ASC "************************"
705:  C42A 2A 20 20          .ASC "*                      *"
706:  C442 2A 20 20          .ASC "*                      *"
707:  C45A 2A 20 20          .ASC "*                      *"
708:  C472 2A 20 20          .ASC "*    ####              *"
709:  C48A 2A 20 20          .ASC "*    ####              *"
710:  C4A2 2A 20 20          .ASC "*    ####              *"
711:  C4BA 2A 20 20          .ASC "*    ####              *"
712:  C4D2 2A 20 20          .ASC "*                      *"
713:  C4EA 2A 20 20          .ASC "*                      *"
714:  C502 2A 20 20          .ASC "*                      *"
715:  C51A 2A 20 20          .ASC "*                      *"
716:  C532 2A 20 20          .ASC "*                      *"
717:  C54A 2A 20 20          .ASC "*                      *"
718:  C562 2A 20 20          .ASC "*                      *"
719:  C57A 2A 20 20          .ASC "*                      *"
720:  C592 2A 20 20          .ASC "*                      *"
721:  C5AA 2A 20 20          .ASC "*                      *"
722:  C5C2 2A 20 20          .ASC "*                      *"
723:  C5DA 2A 20 20          .ASC "*                      *"
724:  C5F2 2A 2A 2A          .ASC "************************"
```

```
                        ;WARTEN AUF TASTENDRUCK
727:   C60A 20 E4 FF KBDWT    JSR  GETIN
728:   C60D C9 00             CMP  #$00
729:   C60F F0 F9             BEQ  KBDWT
730:   C611 60               RTS
'                       ;SPRITE-BENUTZERPROGRAMM
732:   C612 20 E7 C0          JSR  CLAREG   ;ALLE REGISTER LOESCHEN
733:   C615 20 04 C3          JSR  GON      ;GRAFIK EINSCHALTEN
734:   C618 A9 02             LDA  #$02     ;HINTERGRUNDFARBE ...
735:   C61A 20 3D C3          JSR  GCOL     ;BESTIMMEN
736:   C61D 20 2A C3          JSR  GCLR     ;GRAFIKSPEICHER LOESCHEN
737:   C620 A9 02             LDA  #$02     ;SPRITENUMMER
738:   C622 8D 51 C3          STA  SPNUM
739:   C625 A9 07             LDA  #$07     ;SPRITEFARBE
740:   C627 8D 50 C3          STA  SPCOL
741:   C62A 20 9F C3          JSR  SPMAKE   ;SPRITE ERSTELLEN
742:   C62D 12 C4             .WORDQUADER   ;DEFINITIONSTABELLE BEGINNT BEI QUADER
743:   C62F A9 64             LDA  #$64     ;SPRITEKOORDINATEN
744:   C631 85 53             STA  R4       ;X-KOORDINATE
745:   C633 85 55             STA  R5       ;Y-KOORDINATE
746:   C635 20 52 C3          JSR  SPRON    ;SPRITE EINSCHALTEN
747:   C638 20 FD C3          JSR  SPPOS    ;SPRITE POSITIONIEREN
748:   C63B 20 0A C6          JSR  KBDWT    ;WARTEN AUF TASTENDRUCK
749:   C63E 20 68 C3          JSR  SPOFF    ;SPRITE AUSSCHALTEN
750:   C641 20 1B C3          JSR  GOFF     ;GRAFIK AUSSCHALTEN
                        ;
                        ;GRAFIK-BENUTZERPROGRAMM
753:   C644 20 04 C3          JSR  GON      ;GRAFIK EINSCHALTEN
754:   C647 A9 05             LDA  #$05     ;HINTERGRUNDFARBE ...
755:   C649 20 3D C3          JSR  GCOL     ;BESTIMMEN
756:   C64C 20 2A C3          JSR  GCLR     ;GRAFIKSPEICHER LOESCHEN
                        ;
                        ;LINIE (0,0 - 319,199)
759:   C64F 20 00 C0          JSR  ASSI
760:   C652 20 00 00          .BYTE$20,$00,$00,$00  ;XS=0
761:   C656 20 00 C0          JSR  ASSI
762:   C659 0B 01             .BYTE$0B,$01  ;YS=XS
763:   C65B 20 00 C0          JSR  ASSI
764:   C65E 20 02 3F          .BYTE$20,$02,$3F,$01  ;XE=319
765:   C662 20 00 C0          JSR  ASSI
766:   C665 20 03 C7          .BYTE$20,$03,$C7,$00  ;YE=199
767:   C669 20 CE C1          JSR  PLOT     ;LINIE ZEICHNEN
768:   C66C 20 0A C6          JSR  KBDWT    ;WARTEN AUF TASTENDRUCK
769:   C66F 20 1B C3          JSR  GOFF     ;GRAFIK AUSSCHALTEN
770:   C672 60               RTS
```

```
1 REM ..... 16-BIT-ROUTINEN
10 DATA  32,136,192,160,  0,177,163, 41, 32,240
20 DATA  25, 32,151,192, 32,151,192, 32,151,192
30 DATA  32,167,192, 32,158,192, 32,158,192, 32
40 DATA 158,192,160,  3,208, 11, 32,151,192, 32
50 DATA 167,192, 32,158,192,160,  1,177,163,153
60 DATA 251,  0,136, 16,248, 10,168,185, 65,192
70 DATA  72,185, 64,192, 72, 96,230,192,240,192
80 DATA 250,192,  4,193, 20,193, 34,193, 48,193
90 DATA  64,192, 64,192, 64,192, 65,193, 80,193
100 DATA  98,193,118,193, 64,192, 64,192, 64,192
110 DATA  64,192, 64,192, 64,192, 64,192, 64,192
120 DATA  64,192, 64,192, 64,192, 64,192, 64,192
130 DATA  64,192, 64,192, 64,192, 64,192,141,193
140 DATA 153,193,167,193, 64,192,186,232,232,232
150 DATA 189,  0,  1,133,163,232,189,  0,  1,133
160 DATA 164,230,163,208,  2,230,164, 96,165,163
170 DATA 208,  2,198,164,198,163, 96,186,232,232
180 DATA 232,165,163,157,  0,  1,232,165,164,157
190 DATA   0,  1, 96,160,  0,177,163, 72,200,177
200 DATA 163,133,164,104,133,163, 96,165,252, 41
210 DATA 112, 74, 74, 74,170, 96,165,252, 41,  7
220 DATA  10,170, 96, 32,197,192,181, 75,133,163
230 DATA 181, 76,133,164, 96, 32,206,192, 76,216
240 DATA 192,169,  0,162, 16,149, 75,202, 16,251
250 DATA  96, 32,197,192,169,  0,149, 75,149, 76
260 DATA  96, 32,197,192,246, 75,208,  2,246, 76
270 DATA  96, 32,197,192,181, 75,208,  2,214, 76
280 DATA 214, 75,181, 75, 21, 76, 96, 32,197,192
290 DATA 181, 76, 74,149, 76,181, 75,106,149, 75
300 DATA  96, 32,197,192,181, 75, 10,149, 75,181
310 DATA  76, 42,149, 76, 96, 32,197,192, 56,169
320 DATA   0,245, 75,149, 75,169,  0,245, 76,149
330 DATA  76, 96, 32,213,192, 32,206,192,165,163
340 DATA 149, 75,165,164,149, 76, 96, 32,197,192
350 DATA 168, 32,206,192,185, 75,  0,213, 75,185
360 DATA  76,  0,245, 76, 96, 32,225,192, 32,197
370 DATA 192, 24,181, 75,101,163,149, 75,181, 76
380 DATA 101,164,149, 76, 96, 32,206,192,168, 32
```

```
390 DATA 197,192, 56,181, 75,249, 75,  0,149, 75
400 DATA 181, 76,249, 76,  0,149, 76, 96, 32,206
410 DATA 192,165,253,149, 75,165,254,149, 76, 96
420 DATA 165,253,133,163,165,254,133,164, 32,183
430 DATA 192, 76, 69,193, 32,197,192,160,  0,181
440 DATA  75,145,253,200,181, 76,145,253, 96,160
450 DATA   0,145, 75,230, 75,208,  2,230, 76,166
460 DATA  76,228, 78,208,242,166, 75,228, 77,144
470 DATA 236, 96,169,  0,240,  3,  0,169,255,141
480 DATA 210,193, 32,147,194, 32,  0,192, 14, 32
490 DATA  32,  0,192, 14, 49,165, 80, 16, 14,169
500 DATA 255,133, 83,133, 84, 32,  0,192,  7, 32
510 DATA  76,254,193, 32,  0,192, 32,  4,  1,  0
520 DATA 165, 82, 16, 14,169,255,133, 85,133, 86
530 DATA  32,  0,192,  7, 48, 76, 23,194, 32,  0
540 DATA 192, 32,  5,  1,  0, 32,  0,192, 12, 35
550 DATA 176,  3, 76, 94,194,165, 79,208,  4,165
560 DATA  80,240,105, 32,  0,192, 11, 38, 32,  0
570 DATA 192,  5, 96, 32,  0,192, 11, 39, 32,  0
580 DATA 192, 13, 99, 32,  0,192, 12, 98,144, 10
590 DATA  32,  0,192, 14, 98, 32,  0,192, 13, 21
600 DATA  32,  0,192, 13,  4, 32,147,194, 32,  0
610 DATA 192,  4,112,208,219, 96, 32,  0,192, 11
620 DATA  54, 32,  0,192,  5, 96, 32,  0,192, 11
630 DATA  55, 32,  0,192, 13, 98, 32,  0,192, 12
640 DATA  99,144, 10, 32,  0,192, 14, 99, 32,  0
650 DATA 192, 13,  4, 32,  0,192, 13, 21, 32,147
660 DATA 194, 32,  0,192,  4,112,208,219, 96, 44
670 DATA 210,193, 48, 10, 32,174,194,160,  0, 17
680 DATA  91,145, 91, 96, 32,174,194, 73,255,160
690 DATA   0, 49, 91,145, 91, 96,165, 77, 74, 74
700 DATA  74,168,185,231,194, 24,101, 76,133, 92
710 DATA 152, 41,  3,168,165, 75, 41,248,121,  0
720 DATA 195,133, 91,144,  3,230, 92, 24,165, 77
730 DATA  41,  7,101, 91,133, 91,165, 75, 41,  7
740 DATA 168,169,128,192,  0,240,  5, 24,106,136
750 DATA 208,251, 96, 32, 33, 34, 35, 37, 38, 39
760 DATA  40, 42, 43, 44, 45, 47, 48, 49, 50, 52
770 DATA  53, 54, 55, 57, 58, 59, 60, 62,  0, 64
```

```
780 DATA 128,192,173, 17,208,11, 40,195,169, 59
790 DATA 141, 17,208,173, 24,208,141, 41,195,169
800 DATA  24,141, 24,208, 96,173, 40,195,141, 17
810 DATA 208,173, 41,195,141, 24,208, 96,  0,  0
820 DATA  32,  0,192, 32,  0,  0, 32, 32,  0,192
830 DATA  32,  1,255, 63,169,  0, 76,183,193, 72
840 DATA  32,  0,192, 32,  0,  0,  4, 32,  0,192
850 DATA  32,  1,231,  7,104, 76,183,193,  0,  0
860 DATA 173, 81,195, 32,119,195, 13, 21,208,141
870 DATA  21,208,172, 81,195,173, 80,195,153, 39
880 DATA 208, 96,173, 81,195, 32,119,195, 73,255
890 DATA  45, 21,208,141, 21,208, 96,168,169,  1
900 DATA 200,136,240,  3, 10,208,250, 96,  3,  3
910 DATA   3, 64,128,192, 13, 14, 15,172, 81,195
920 DATA 185,136,195,153,248,  7,185,133,195,133
930 DATA  77,185,130,195,133, 78, 96, 32,136,192
940 DATA  32,151,192, 32,167,192, 32,158,192, 32
950 DATA 183,192,162,  0, 32, 72,193, 32,139,195
960 DATA 160,  0,169, 21,133, 79,162, 24,169,  8
970 DATA 133, 81,132, 82,177, 75,230, 75,208,  2
980 DATA 230, 76, 41,127,201, 32,240,  3, 56,176
990 DATA   1, 24, 38, 82,202,240, 17,198, 81,208
1000 DATA 229,165, 82,145, 77,230, 77,208,  2,230
1010 DATA  78, 76,190,195,198, 81,165, 82,145, 77
1020 DATA 230, 77,208,  2,230, 78,198, 79,208,192
1030 DATA  96,173, 81,195, 10,168,165, 85,153,  1
1040 DATA 208,165, 83,153,  0,208,165, 84,141, 16
1050 DATA 208, 96, 42, 42, 42, 42, 42, 42, 42, 42
1060 DATA  42, 42, 42, 42, 42, 42, 42, 42, 42, 42
1070 DATA  42, 42, 42, 42, 42, 42, 42, 32, 32, 32
1080 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1090 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 42
1100 DATA  42, 32, 32, 32, 32, 32, 32, 32, 32, 32
1110 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1120 DATA  32, 32, 32, 42, 42, 32, 32, 32, 32, 32
1130 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1140 DATA  32, 32, 32, 32, 32, 32, 32, 42, 42, 32
1150 DATA  32, 32, 32, 42, 42, 42, 42, 32, 32, 32
1160 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
```

```
1170 DATA  32, 42, 42, 32, 32, 32, 32, 42, 42, 42
1180 DATA  42, 32, 32, 32, 32, 32, 32, 32, 32, 32
1190 DATA  32, 32, 32, 32, 32, 42, 42, 32, 32, 32
1200 DATA  32, 42, 42, 42, 42, 32, 32, 32, 32, 32
1210 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 42
1220 DATA  42, 32, 32, 32, 32, 42, 42, 42, 42, 32
1230 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1240 DATA  32, 32, 32, 42, 42, 32, 32, 32, 32, 32
1250 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1260 DATA  32, 32, 32, 32, 32, 32, 32, 42, 42, 32
1270 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1280 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1290 DATA  32, 42, 42, 32, 32, 32, 32, 32, 32, 32
1300 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1310 DATA  32, 32, 32, 32, 32, 42, 42, 32, 32, 32
1320 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1330 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 42
1340 DATA  42, 32, 32, 32, 32, 32, 32, 32, 32, 32
1350 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1360 DATA  32, 32, 32, 42, 42, 32, 32, 32, 32, 32
1370 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1380 DATA  32, 32, 32, 32, 32, 32, 32, 42, 42, 32
1390 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1400 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1410 DATA  32, 42, 42, 32, 32, 32, 32, 32, 32, 32
1420 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1430 DATA  32, 32, 32, 32, 32, 42, 42, 32, 32, 32
1440 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1450 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 42
1460 DATA  42, 32, 32, 32, 32, 32, 32, 32, 32, 32
1470 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1480 DATA  32, 32, 32, 42, 42, 32, 32, 32, 32, 32
1490 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1500 DATA  32, 32, 32, 32, 32, 32, 32, 42, 42, 32
1510 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1520 DATA  32, 32, 32, 32, 32, 32, 32, 32, 32, 32
1530 DATA  32, 42, 42, 42, 42, 42, 42, 42, 42, 42
1540 DATA  42, 42, 42, 42, 42, 42, 42, 42, 42, 42
1550 DATA  42, 42, 42, 42, 42, 42, 32,228,255,201
```

```
1560 DATA   0,240,249, 96, 32,231,192, 32,  4,195
1570 DATA 169,  2, 32, 61,195, 32, 42,195,169,  2
1580 DATA 141, 81,195,169,  7,141, 80,195, 32,159
1590 DATA 195, 18,196,169,100,133, 83,133, 85, 32
1600 DATA  82,195, 32,253,195, 32, 10,198, 32,104
1610 DATA 195, 32, 27,195, 32,  4,195,169,  5, 32
1620 DATA  61,195, 32, 42,195, 32,  0,192, 32,  0
1630 DATA   0,  0, 32,  0,192, 11,  1, 32,  0,192
1640 DATA  32,  2, 63,  1, 32,  0,192, 32,  3,199
1650 DATA   0, 32,206,193, 32, 10,198, 32, 27,195
1660 DATA  96
1670 FOR I = 49152 TO 50802
1680 READ K: POKE I,K
1690 SUM=SUM+K
1700 NEXT I
1710 IF SUM <> 145069 THEN PRINT "PRUEFSUMME FALSCH!": END
1720 PRINT "PRUEFSUMME RICHTIG!"
```

Literaturverzeichnis

COMMODORE BUSINESS MACHINES, INC. (Hrsg.): "Commodore 64 Programmer's Reference Guide". HOWARD W. Sams u. Co., Inc., Indianapolis (1983).

Sehr umfangreiche (englische) Dokumentation zum Commodore 64. Inzwischen gibt es auch eine deutsche Version.

COMMODORE: COMMODORE PROGRAMMIER-HANDBUCH.

Detaillierte technische Beschreibung des mit dem 6510 kompatiblen Mikroprozessors 6502.

ZAKS, RODNAY: Programmierung des 6502. Sybex (1982).

Ein "Standardwerk" zum 6502 ohne Bezug auf einen bestimmten Computer.

Register

16-Bit-Interpreter 101 ff.
16-Bit-Register 15, 77 f.
16-Bit-Simulationen 77 ff.
6510-Befehle
- alphabetische Übersicht 172 ff.
- alphabetische Kurzübersicht 169 ff.
- hexadezimale Reihenfolge 201 ff.
- Klassifikation 205
6510 Mikroprozessor 15 ff.

A

ADC 44
Adresse, symbolische 50
Adressierung
- direkte 49
- indizierte 49
Adressierungsart
- alphabetische Übersicht 172 ff.

nicht indiziert:
- implizit 161
- unmittelbar 161 f.
- absolut 162
- Zero Page 162
- relativ 162 f.

indiziert:
- absolut indiziert 164
- Zero Page indiziert 164
- indiziert-indirekt 165
- indirekt-indiziert 165
- indirekt absolut 166

Akkumulator 16
Algorithmus 21
AND 102
ASL 87

B

BASIC-Ladeprogramm-Generator 214 f.
BCC 52
BCS 122
BEQ 102
Binär-System 29 ff.
BIT 118
Bit 29
BMI 118
BNE 50
BPL 88
BRK 45
BVC 155
BVS 155
Byte 29,35

C

CLC 44
CLD 156
CLI 157
CLV 155
CMP 95
Compiler 11
CPU 11
CPX 50
CPY 155

D

DEC 83
DEX 88
DEY 104
Dezimalbetrieb 156
Disassembler-Programm 206 ff.

E

Einerkomplement 31
EOR 119

F

Flags 17 f.
Flußdiagramm 22 ff.

G

Grafik 109 ff.
Grafik-Speicher 114

H

Hexadezimalcode-Eingabe-
  Programm 206 ff.
Hexadezimal-System 35 ff.

I

INC 82
Index-Register 16
integrierte Schaltung 11
Interpreter 11
Interrupt 156
INX 50
INY 52

J

JMP 88
JSR 61

K

Kassettenpuffer 77

L

Label 50
LDA 44
LDX 49
LDY 65
LIFO 16
LSR 86

M

Maschinensprache 11
Mikroprozessor 11
  - schematische Dar-
    stellung 15
Mnemonic 43

N

NOP 158

O

ORA 90

P

Parameterübergabe 64 ff.
  - von BASIC aus 143 f.
PHA 67
PHP 155
PLA 67
PLP 155
Programmunterbrechung 156
Programmzähler 16
Prozessor-Status-Register
  17

R

Register 16,77
RTI 157
RTS 62
Rücksprungadresse 67 f., 80
  ff.,84,103 ff.
ROL 93
ROM-Routinen 143 ff.
ROR 91

S

SBC 94
SEC 94
SED 156
SEI 157
Sprites 126 ff.
  - Farbspeicher 137
STA 44
Stack 16,67 ff.
Stapelzeiger 16
STX 54
STY 52
Subroutine 61

T

TAX 73
TAY 52
TSX 63
TXA 50
TXS 63
TYA 66

U

Überlauf 32 ff.
Übertrag 32 ff.
Unterprogramm 25,61 ff.

Z

Zahlensysteme 29 ff.
Zentraleinheit 11
Zero Page 48
Zweierkomplement 31T

Dieses Buch bietet eine systematische Einführung in die Programmiersprache BASIC. Außer vielen kleineren Programmen zur Illustrierung der BASIC-Anweisungen gibt es eine umfangreiche Programmsammlung zu den verschiedensten Themenbereichen. Die besonderen Fähigkeiten des C 64 werden mit vielen Programmbeispielen erläutert.

1983. 356 Seiten.
Spiralh. DM 56,–/Fr. 56.–
ISBN 3-88322-029-9

Der C 64 bietet vielseitige grafische Möglichkeiten. Dieses Buch gibt Informationen wie man Grafikfunktionen anwendet. Ausgehend von Grafiken mit den ›festen‹ Grafik-Zeichen wird systematisch zu den anspruchsvolleren Möglichkeiten, illustriert durch typische Beispiele, geführt.

1983. 138 Seiten, 1 Folie.
Spiralh. DM 38,–/Fr. 38.–
ISBN 3-88322-027-2

Bekanntlich verfügt der C 64 ›von Haus aus‹ über einen Baustein, der die Erzeugung von mehrstimmiger Musik erlaubt. Sowohl der Anfänger ohne musikalische Vorkenntnisse wird angesprochen, als auch der Musiker, der seine Ideen mit Hilfe des Computers umsetzen möchte.

1984. 312 Seiten.
Spiralh. DM 48,–/Fr. 48.–
ISBN 3-88322-046-9

Dieses Buch enthält eine ganze Reihe von sofort lauffähigen Spiel- und Simulationsprogrammen, möchte aber auch dazu anregen, diese Programme zu verändern und weiterzuentwickeln. Besonders reizvoll dürfte es wohl sein, den ›lernenden‹ Programmen noch etwas mehr ›Intelligenz‹ zu verleihen.

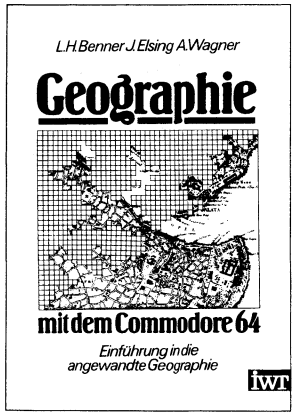
1984. 208 Seiten.
Spiralh. DM 38,–/Fr. 38.–
ISBN 3-88322-050-7

Grafikprogramme werden ›gehirngerecht‹ aufbereitet, d.h. man sieht, wie Grafikbefehle ›gehen‹: Neue Art des Formats – man bekommt ein ›Bild‹ des Befehls · Demo-Programme unterstützen das Gedächtnis · Bildschirm-Hardcopies als schnelles Nachschlagewerk · farbige Übersichtskarten zur Programmier-Erleichterung.

1984. 208 Seiten
Spiralh. DM 44,–/Fr. 44.–
ISBN 3-88322-056-6

Die Programmierung des neuen Video Interface Chips 6567 ist das Hauptthema des Buches. Basic-Grafikprogramme werden von Maschinenprogrammen zum Punktsetzen und Linienzeichnen unterstützt, was die Schnelligkeit um ein Vielfaches erhöht. Teilweise werden Basic-Programme direkt in Maschinensprache parallel dargestellt.

1984. 152 Seiten.
Spiralh. DM 38,–/Fr. 38.–
ISBN 3-88322-051-5



Dieses Buch führt den Leser in die verschiedenen Sparten der Geographie ein. Die Klimatologie, Hydrologie, Geomorphologie und Bodenkunde werden ebenso wie die Kulturgeographie, Verkehrsgeographie und Demographie ausführlich behandelt.

1984. Ca. 160 Seiten.
Spiralh. Ca. DM 38,–/ca. Fr. 38.–
ISBN 3-88322-057-4

Das Buch führt den Leser mit vielen Beispielen aus der Praxis in das Datenbanksystem dBase II ein. Es sind keine Vorkenntnisse von dBase erforderlich. Das Werk ist ähnlich einer (PU) Programmierten Unterweisung aufgebaut und ist daher sehr gut verständlich.

1984. Ca. 250 Seiten.
Geb. Ca. DM 58,–/ca. Fr. 58.–
ISBN 3-88322-136-8

Dieses Buch zeigt, wie sich komplizierte Operationen verständlich beschreiben lassen. Es wird demonstriert, wie einfach sich dreidimensionale Probleme lösen lassen. Die Beispiele reichen von der Geraden über das Dreikörperproblem bis hin zum dreidimensionalen Planetensystem.

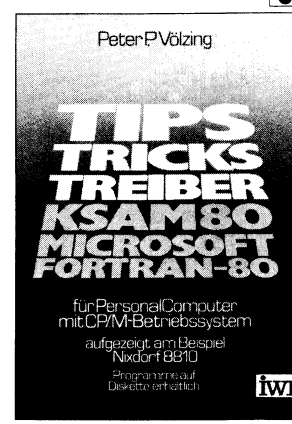
1984. 208 Seiten.
Kart. DM 44,–/Fr. 44.–
ISBN 3-88322-052-3

Dieses Buch behandelt die Problematik der Schnittstellen V 24 und dem IEC-Bus. Die Übertragung zu peripheren Geräten (Drucker, Plotter, Terminal u. a.) ist gut verständlich aufgezeigt. Hardwareaufbau und Beispielanschlüsse machen das Buch zu einem guten Nachschlagewerk.

1984. Ca. 200 Seiten.
Geb. Ca. DM 48,–/ca. Fr. 48.–
ISBN 3-88322-094-9

Dieses Buch führt den Leser mit vielen Beispielen in die Programmier-Sprache Cobol ein. Alle Sprachelemente werden ausführlich, detailliert und verständlich erklärt. Dieses Werk ist sowohl als Lehrbuch wie auch als Nachschlagewerk konzipiert.

1984. Ca. 220 Seiten.
Geb. DM 56,–/Fr. 56.–
ISBN 3-88322-133-3



Dieses Buch vermittelt dem Leser Tips und Tricks zur optimalen Fortran-Programmierung. Schwerpunkte bilden u. a. Bildschirmtreiber, Druckersteuerung und Indexdateibehandlung. Das KSAM 80-Modul wird an praktischen Beispielen erläutert.

1984. 392 Seiten.
Spiralh. DM 56,–/Fr. 56.–
ISBN 3-88322-122-8

Eine Hilfestellung für wirtschaftliche Entscheidungen sind Programmsammlungen, die die guten Grafik- und Farbmöglichkeiten des Computers nutzen. Diagramme, Sprites, optische Darstellungen von Simulationen werden eingesetzt, die die Ergebnisse verdeutlichen. Die finanzmathematischen Grundlagen sind zu jedem Programm beschrieben.

1983. 224 Seiten. Mit mehr. Abb.
Spiralh. DM 38,–/Fr. 38.–
ISBN 3-88322-030-2

Dieses Buch hilft den Schülern der Oberstufe beim Lösen und Bearbeiten ihrer Hausaufgaben. Es bietet unter anderem eine Hilfestellung beim Differenzieren, Integrieren von Funktionen, bei Kurvendiskussionen und beim Lösen von Gleichungssystemen.

1984. 148 Seiten.
Kart. DM 32,–/Fr. 32.–
ISBN 3-88322-125-2

Dieses Buch enthält 40 mathematische Programme aus den Bereichen: Mehrregister-Arithmetik – Zahlentheorie – Kombinatorik – Algebra – Geometrie – numerische Mathematik. Neu ist die Langzahl-Arithmetik. Sie gestattet die Grundrechenarten für Zahlen bis 255 Stellen.

1984. 260 Seiten.
Kart. DM 42,–/Fr. 42.–
ISBN 3-88322-048-5

Dieses Buch behandelt den Einsatz des deutschsprachigen IWT LOGO auf dem C 64. Anhand vieler ausführlicher Beispiele aus Mathematik, Geometrie, Grammatik und Musik wird der Einsatz von IWT LOGO im Unterricht erläutert.

1984. 200 Seiten.
Spiralh. DM 44,–/Fr. 44.–
ISBN 3-88322-058-2

Zeichenerklärung

Viele Programme, die in den Büchern abgedruckt sind, erhalten Sie auch auf ● Diskette oder ⌀ Kassette. Achten Sie auf die Kennzeichnung bei den Büchern, genaue Beschreibung in unserem Software-Katalog.

Allgemeines

Unsere gebundenen Preise verstehen sich incl. ges. MwSt. zuzügl. Versandspesen. Stand: 1. 3. 1984, Änderungen und Irrtum vorbehalten. Es gelten ausschließlich unsere Liefer- und Zahlungsbedingungen vom Februar 1984.
IWT-Bücher erhalten Sie in jeder guten Buchhandlung, im Computer-Fachhandel, in Kaufhäusern und direkt beim Verlag.

IWT Verlag GmbH, Wendelsteinstraße 3, Postfach 11 44, 8011 Vaterstetten, Tel. (0 81 06) 3 10 17, Telex 5213989 iwt

IWT Software Service GmbH, Höhestraße 66, Postfach 13 25, 5093 Burscheid 1, Tel. (0 21 74) 6 28 15, Telex 5213989 iwt

H. Sterner/T. Tutughamiarsa

Maschinensprache auf dem Commodore 64

Das vorliegende Buch ist in seiner Konzeption an den Bedürfnissen des „Einsteigers" orientiert: offensichtlich ist es wenig sinnvoll, den Befehlssatz des 6510-Prozessors ohne Anwendungsbeispiele zunächst nur „auf dem Trockenen" zu besprechen. Deshalb wird ein Befehl dann eingeführt und erklärt, wenn er für eine Problemlösung erforderlich ist. Die Erarbeitung des Befehlssatzes erfolgt also aus den jeweiligen praktischen Anforderungen heraus.
Den zentralen Teil des Buches bildet ein Paket von 16-Bit-Routinen, die z. B. für die Grafik-Programmierung genutzt werden können.
Weiterhin werden einige häufig verwendete interne ROM-Routinen des COMMODORE 64 beschrieben, die der Benutzer auch in seine eigenen Programme einbauen kann.
Den Abschluß des Buches bildet eine Übersicht über die Adressierungsarten des 6510.

ISBN 3-88322-047-7