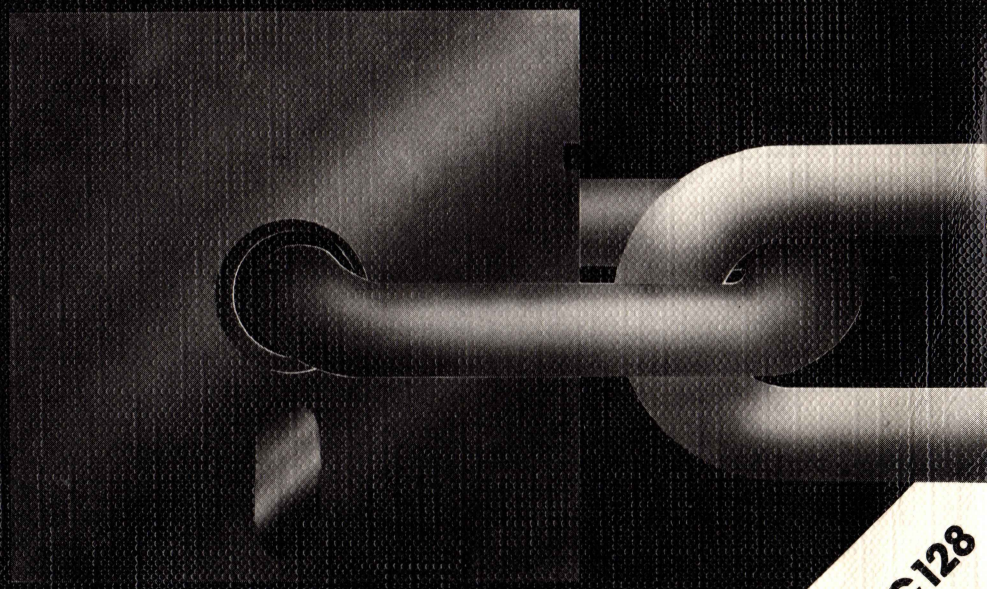
DAS ANTI-CRACKER BUCH

Gelfand
Felt
Strauch
Krsnik

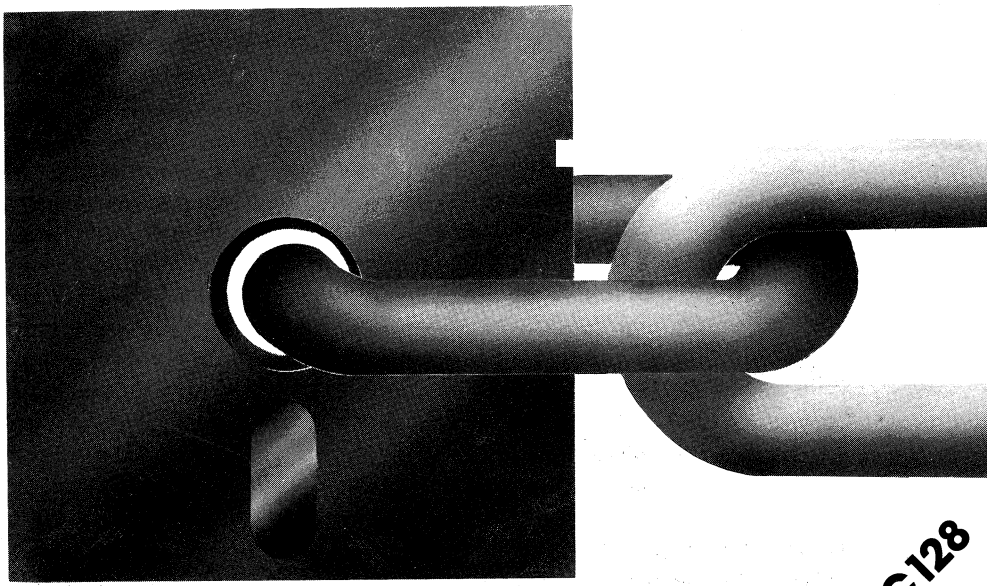


Für C64 + C128

Ein DATA BECKER Buch

DAS ANTI-CRACKER BUCH

Gelfand
Felt
Strauch
Krsnik



Für C64 + C128

Ein DATA BECKER Buch

ISBN 3-89011-253-6

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Zu diesem Buch

Da dieses Buch sich mit einem recht heiklen Thema beschäftigt, dessen Brisanz vor allem in letzter Zeit immer mehr ansteigt, hielten wir es für notwendig, einige einleitende Worte zu diesem Thema zu schreiben.

Während der gesamten Entstehungszeit dieses Buches haben wir uns immer an dem Ziel orientiert, möglichst jedem C64-Benutzer einen Einblick in die aktuellsten Schutz- bzw. Kopierschutzsysteme zu ermöglichen. Wir versuchen hier, unser Wissen über diese Systeme zu vermitteln, um Ihnen die Möglichkeit zu geben, Ihre eigenen Programme vor Fremdzugriffen und Raubkopierern zu schützen. Darüber hinaus haben wir neue, geradezu sensationelle Kopierschutzverfahren entdeckt und entwickelt, die von keinem derzeit bekannten Kopierprogramm überwunden werden können.

Da wir versucht haben, alle zur Zeit auf dem Markt befindlichen Techniken zusammenzustellen und diese zu erklären, ist es Ihnen mit diesem Buch auch möglich, fremde Software zu untersuchen und zu verstehen. Das soll Sie aber nicht dazu animieren, diese Software in irgendeiner Art zu modifizieren.

Ausdrücklich warnen (!) wir davor, den Kopierschutz von fremder Software zu entfernen und diese zu kopieren.

In diesem Sinne wünschen wir Ihnen viel Spaß und vor allem viel Erfolg!

Die Autoren

Düsseldorf,
im Februar 1987

Inhaltsverzeichnis

1. Einleitung

Der C64 ist ein Computer, für den tausende von Programmen zur Verfügung stehen. Man kann diese Software einerseits im Laden kaufen, andererseits ist nahezu jedes Programm auch als Raubkopie auf dem schwarzen Markt erhältlich. Dieses Potential hat zwar zur weiten Verbreitung des Rechners geführt, aber man kann heute kein Programm mehr auf den Markt bringen, befürchten zu müssen, daß es binnen einer Woche als Raubkopie vorliegt und somit sehr viel seltener verkauft wird. Aus diesem Grunde werden die meisten Programme von den Firmen mit einem Kopierschutz ausgestattet.

Wir wollen in diesem Buch kein Urteil über das Raubkopieren abgeben. Vielmehr wollen wir aufzeigen, wie man sein Programm vor den unberechtigten Zugriffen anderer schützt und welche Kopierschutzmethoden auf dem C64 möglich sind. Die Palette der von uns beschriebenen Systeme reicht von einfachen, aber nichtsdestoweniger wirkungsvollen, Tricks zum Schutz eines BASIC-Programmes bis zu Anwendungen, die die von professionellen Entwicklern um einiges übertreffen.

Als Programmierer können Sie durch dieses Buch einen Großteil der Arbeit, die Sie sonst auf den Kopierschutz verwenden müßten, einsparen. Sie können allerdings auch als Heimanwender Ihre Programme vor dem Zugriff durch Bekannte schützen.

Wir haben uns bemüht, unsere Beschreibungen auch für einen Anfänger verständlich zu halten. Die meisten abgedruckten Programme können Sie benutzen, ohne sich bis ins Detail mit ihnen zu beschäftigen. Andererseits wollten wir natürlich auch dem erfahrenen Anwender etwas bieten. Daher sind alle Programm-Listings, insbesondere die Maschinensprache-Listings, ausführlich erläutert und dokumentiert. Wenn Sie, aufbauend auf den Beispielen, zu Eigenentwicklungen übergehen wollen, kann es an einigen Stellen hilfreich sein, schon mit der 6510-Assemblersprache vertraut zu sein oder sich zumindest mit der Bedienung eines Maschinensprachemonitors ein wenig auszukennen. Wie gesagt, ist das allerdings keine Grundvoraussetzung, um mit diesem Buch etwas anzufangen.

Um einen Kopierschutz zu überwinden, kann ein Kopierer prinzipiell zwei Dinge tun. Entweder versucht er, das Programm mit Hilfe eines Kopierprogrammes zu vervielfältigen, oder er entfernt die Abfrage des Kopierschutzes. Daher mußten wir in diesem Buch auch eine klare Grenze zwischen Programm- und Kopierschutz ziehen. Kopierschutz ist alles, was das Kopieren eines Programms verhindert. Ein Programmschutz dagegen dient dazu, Änderungen an einem Programm zu verhindern, worunter natürlich auch das "Knacken", also das Entfernen eines Kopierschutzes, fällt.

Beim Kopierschutz mußte eine weitere Unterteilung in Datasetten- und Disketten-Kopierschutz vorgenommen werden. Die Möglichkeiten, die eine Diskettenstation bietet, unterscheiden sich stark von denen eines Datasetten-Rekorders. Auch beim Programmschutz haben wir eine Unterscheidung getroffen, nämlich zwischen BASIC- und Assembler-Programmen. Für beide Fälle finden Sie in diesem Buch zahlreiche Beispiele.

In neuerer Zeit ist eine Kopiermethode sehr beliebt geworden, die einer gesonderten Behandlung bedarf: das "Knacken" mit einem "Knackmodul". Selbstverständlich haben wir auch für dieses Schutzproblem eine Lösung gefunden.

Betrachten Sie dieses Buch nicht nur als eine Ansammlung aller möglichen Schutzsysteme, sondern lassen Sie sich auch dazu anregen, die angeführten Beispiele zu erweitern und eigene Ideen zu verwirklichen.

2. BASIC-Programmschutz

2.1 Listschutz

Wir werden hier einige Methoden demonstrieren, wie man sein eigenes BASIC-Programm vor dem LIST-Befehl schützt. Alle diese Möglichkeiten, die wir Ihnen hier aufführen, können natürlich mit anderen Schutz-Methoden kombiniert werden. Zum Beispiel ein BASIC-Programm, das einen LIST-Schutz enthält, der nur ein Listen der Zeilennummern ermöglicht, RUN-STOP-RESTORE ignoriert und nach RESET ein automatisches Starten des Listings oder ein Zerstören des Programms veranlaßt.

Wir werden zunächst mit den einfachsten Verfahren beginnen und gehen anschließend zu den etwas anspruchsvolleren über. Wir hoffen, alle BASIC-LIST-Schutzmethoden erfaßt zu haben.

Wir beginnen zunächst mit dem LIST-Schutz. Anschließend folgen dann die anderen Schutz-Möglichkeiten wie RESET-Schutz usw.

2.1.1 Der Listschutz mit "SHIFT-L"

Wir wollen Ihnen hier eine einfache Methode demonstrieren, mit der Sie BASIC-Programme vor dem Listen schützen können. Diesen LIST-Schutz zu installieren, ist nicht sehr schwer. Man muß hinter einer BASIC-Zeile nur noch einen zusätzlichen Befehl eingeben, und zwar ein REM hinter dem direkt noch ein [SHIFT]-L folgt. Dies könnte folgendermaßen aussehen:

```
10 FOR I=1 TO 20:REM[SHIFT]L
```

Mit [SHIFT]-L ist das Zeichen gemeint, das durch gleichzeitiges Drücken der SHIFT- und der L-Taste erzeugt wird. Dieses Zeichen muß direkt hinter dem REM-Befehl folgen. Falls man nun versucht, das Programm zu listen, wird ab dieser Zeile die Fehlermeldung "SYNTAX ERROR" folgen, und der List-Vorgang wird unterbrochen.

Der Nachteil bei diesem Schutz ist allerdings, daß er nur auf dem Original-Betriebssystem funktioniert. Außerdem wird die Zeile 10, in der sich das REML befindet, zuerst gelistet, und dann erst folgt die Fehlermeldung. Das heißt, daß die erste Zeile immer sichtbar bleibt. Also ist es ratsam, vor dem eigentlichen Programm noch ein paar REM-Zeilen zu setzen, damit man den Programmbeginn nicht sieht:

```
5 REML
6 REML
7 REML
10 FOR I=1 TO 20:REML
.
.
.
```

Beim Versuch das Programm zu listen, sieht man nur folgendes:

```
5 REM
?SYNTAX ERROR

READY.
```

Das restliche Programm bleibt unsichtbar. Noch ein Hinweis: Dieser LIST-Schutz läßt sich sehr leicht wieder ausbauen, indem man die REM-Zeilen nacheinander löscht. So kann man sich langsam, aber sicher in das Programm hineintasten. Möchten Sie ein Programm effizient schützen, ist dieser Schutz nicht zu empfehlen.

2.1.2 LIST-Schutz mit Steuerzeichen

Dieser LIST-Schutz funktioniert ähnlich wie der schon beschriebene. Der Unterschied besteht nur darin, daß statt eines [SHIFT]-L Steuerzeichen eingesetzt werden. Für den LIST-Schutz eignet sich eigentlich nur das DEL-Zeichen. Es hat die gleiche Funktion wie die DELETE-Taste auf der Tastatur. Wenn

man nun das Zeichen in eine BASIC-Zeile mit einem REM-Befehl kombiniert, lassen sich BASIC-Zeilen verstümmeln bzw. ganz unsichtbar machen:

```
10 FOR I=1 TO 20:REM"TTTTTTTTTTTTTTTT
```

Beim Listen wird für jedes reverse T ein Zeichen der BASIC-Zeile gelöscht. Bei unserem Beispiel wird bei einem LIST-Vorgang alles bis auf die Zeilennummer gelöscht. Ist dieser Schutz einmal eingebaut, läßt er sich schlecht wieder ausbauen. Um ein solches reverses T zu erzeugen, muß wie folgt vorgegangen werden:

```
10 FOR I=1 TO 20:REM""[DEL]17 MAL [INS] UND 17 MAL [DEL]
```

Sofort hinter dem REM werden zwei Hochkommata gesetzt. Anschließend wird eines davon mit der DELETE-TASTE wieder entfernt, um den Hochkomma-Modus auszuschalten. Dann drücken Sie 17mal die INSERT-TASTE und anschließend 17mal die DELETE-TASTE. Jetzt haben Sie 17 reverse T erzeugt. Die Anzahl kann je nach Bedarf gewählt werden.

Wenn Sie über SPEEDDOS oder ein anderes System verfügen und dieses auch benutzen, ist es durchaus möglich, daß diese Schutz-Methode nicht funktioniert, weil diese Systeme über eine veränderte LIST-Routine verfügen.

2.1.3 LIST-Schutz über Sprungvektoren

Einen guten LIST-Schutz kann man auch durch Ändern des LIST-Vektors erreichen. Dieser Vektor, der in $0306 (774) und $0307 (775) liegt, wird bei jeden LIST-Befehl angesprungen, um das BASIC-Programm in Klartext zu verwandeln. Er dient nur als Zeiger in LOW- und HIGH-Byte-Darstellung, der auf diese Betriebssystem-Routine zeigt.

Man kann nun diesen Vektor mittels POKE oder über einen Maschinensprachemonitor auf eine andere Routine verzweigen lassen. Das heißt, daß bei jedem LIST-Befehl diese Routine angesprungen wird.

Geben Sie nun einmal diese beiden POKEs ein:

```
POKE 774,226: POKE 775,252
```

Mit diesen POKEs haben Sie den LIST-Vektor auf die RESET-Routine $FCE2 (64738) verzweigen lassen. Wenn man nun LIST eingibt, wird automatisch ein RESET ausgeführt. Das funktioniert aber nur, wenn Sie ein BASIC-Programm im Speicher haben. Auf diese Weise kann man auch zu seiner eigenen Maschinensprache-Routine verzweigen, wo zum Beispiel das BASIC-Programm automatisch wieder gestartet oder zerstört werden kann.

Einen kleinen Nachteil hat dieses Verfahren. Man muß immer nach dem Einschalten des Computers diese POKEs eingeben. Dieses Problem läßt sich lösen, indem man einen Autostart in das Programm einbaut, der diese Vektoren automatisch umstellt.

2.1.4 Nur Zeilennummern

Dieser LIST-Schutz eignet sich am besten zum Schutz von BASIC-Programmen. Wenn man hier bei eingebautem LIST-Schutz versucht, das BASIC-Programm zu listen, sieht man nur die jeweiligen Zeilennummern der BASIC-Zeilen. Der Nachteil ist, daß man jede BASIC-Zeile einzeln mit dem Schutz versehen muß.

Nun zum eigentlichen Schutz! Geben Sie folgende Zeile ein:

```
10 :::::PRINT "HALLO"
```

Diese fünf Doppelpunkte dienen als Platzhalter. Sie werden später mit dem Maschinensprachemonitor überschrieben. Aber

schauen wir uns erst einmal diesen BASIC-Befehl im Maschinensprachemonitor an:

```
.:0800  00 13 08 0A 00 3A 3A 3A   .....:::
.:0808  3A 3A 99 22 48 41 4C 4C   ::."HALL
.:0810  4F 22 00 00 00 . . .      O".....
```

Die ersten beiden Werte in Zeile $0800 zeigen in LOW- und HIGH-Byte-Darstellung auf die nächste BASIC-Zeile. Falls keine weitere BASIC-Zeile vorhanden ist, weist dieser Zeiger auf drei hintereinander folgende Nullen, die das BASIC-Programmende markieren. Dieser Zeiger heißt Linker.

Die nächsten beiden Werte stellen auch in LOW- und HIGH-Byte-Darstellung die Zeilennummer dar. Im LOW-Byte steht $0A und im HIGH-Byte $00, das heißt, daß in diesem Fall die Zeilennummer 10 beträgt. Eine LOW-HIGH-Byte-Darstellung ist unbedingt erforderlich, da man sonst nur Zeilennummern bis 255 darstellen könnte. In diesem Fall aber lassen sich Zeilennummern bis 65535 darstellen.

Anschließend folgen unsere fünf Doppelpunkte, die, wie schon erwähnt, nur als Platzhalter dienen. Sie sind im Maschinensprachemonitor mit dem ASCII-Wert $3F abgelegt.

Hinter den Doppelpunkten folgt dann das Token $99, was nichts anderes als PRINT im BASIC-Listing bedeutet. Für jeden BASIC-Befehl ist ein bestimmtes Token vorhanden.

Die ASCII-Zeichen hinter dem Token sind die Zeichen, die über dem PRINT-Befehl ausgegeben werden sollen.

Falls eine weitere BASIC-Zeile existiert, würde dann $00 folgen, und der Vorgang würde sich wiederholen. Die Null signalisiert, daß eine neue BASIC-Zeile folgt. Wenn aber keine weitere BASIC-Zeile folgt, wie es in unserem Beispiel der Fall ist, stehen dort drei Nullen, die das BASIC-Programmende anzeigen.

Soweit zum Aufbau der BASIC-Zeile. Nun wollen wir uns aber mit dem eigentlichen LIST-Schutz beschäftigen. Wir ändern die Doppelpunkte, also die Platzhalter, mit dem Maschinensprachemonitor:

```
.: 0800 00 13 08 0A 00 00 FF FF   ........
.: 0808 FF FF 99 22 48 41 4C 4C   ...."HALL
.: 0810 4F 22 00 00 00 . . .      O".....
```

Anstelle der Doppelpunkte $3A $3A $3A $3A $3A haben wir jetzt $00 $FF $FF $FF $FF gesetzt. Wenn wir nun LIST eingeben, sehen wir:

```
10

READY.
```

Die Zeilennummer bleibt sichtbar, aber der BASIC-Befehl ist verschwunden. Er ist natürlich nicht richtig "verschwunden", sondern er läßt sich einfach nicht mehr listen, weil der Interpreter direkt hinter der Zeilennummer eine Null findet, diese als Zeilenende interpretiert und zur nächsten Zeile springt.

Das Programm aber läßt sich immer noch ganz normal mit RUN starten. Der LIST-Schutz funktioniert auch, wenn mehrere BASIC-Befehle in einer Zeilen stehen. Da es sicher umständlich ist, jede BASIC-Zeile einzeln zu schützen, haben wir zwei Programme für Sie vorbereitet:

Das erste ist ein Packer-Programm für BASIC-Zeilen. Es packt bis zu 245 Zeichen pro Zeile. Programmzeilen, die von GOTO, GOSUB oder THEN angesprungen werden, werden nicht gebunden. Auch Zeilen, in denen REM-Zeilen vorkommen, bleiben unverändert, weil sonst der nachfolgende Befehl als REM-Text anerkannt werden würde. Alle PRINT- und OPEN-Befehle müssen mit Anführungszeichen abgeschlossen sein, da sonst vom Compaktor "SYNTAX ERROR" ausgegeben wird.

Wenn man nun ein BASIC-Programm packen möchte, muß man den Packer vorher in den Speicher laden. Das Packer-Programm befindet sich nun ab der Adresse $C000 (49152) im Speicher.

Danach muß ein NEW eingegeben werden, um die Zeiger wieder richtig zu stellen. Anschließend muß das zu packende BASIC-Programm in den Speicher geladen werden.

Das Packerprogramm wird mit SYS 49152 gestartet. Der Pack-Vorgang dauert ca. 3 Sekunden. Das BASIC-Programm kann nun wieder auf Diskette gespeichert werden. Es ist um einige Bytes kürzer als vorher.

Das zweite Programm, das direkt hinter dem Packerprogramm im Speicher liegt, ist das Listschutz-Programm. Es kann "normale" und gepackte BASIC-Programme schützen.

Wenn sich das zu schützende BASIC-Programm im Speicher befindet, kann die Schutz-Routine mit SYS 49730 ($C247) aufgerufen werden. Nach ca. einer Sekunde sind alle BASIC-Zeilen geschützt, das heißt, es sind nur die Zeilennummern sichtbar.

Das BASIC-Programm kann auch normal mit SAVE wieder auf Diskette oder Kassette gespeichert werden. Es ist durch den Schutz um einiges länger als vorher. Wenn man aber das BASIC-Programm vorher packt, kommt man ungefähr wieder auf die gleiche Länge. Ein gepacktes BASIC-Programm ist auch schwer zu ändern, daher kann man das Packen auch als Änderungsschutz verwenden.

Vorsicht! Wenn Sie einmal ein BASIC-Programm LIST-geschützt haben, wird es kaum möglich sein, es wieder sichtbar zu machen.

Nun das Maschinenlisting des Packer-Programms:

```
C000 LDA #$36       BASIC-Interpreter
C002 STA $01        ausblenden
C004 JSR $C0A7      Tabelle angesprungen. Zeilen erstellen
C007 LDA $2B        BASIC-Anfang LOW-Byte
C009 SEC            Carry für Subtraktion setzen
C00A SBC #$01       minus Eins
C00C STA $AB        nach Zeiger für Zielspeicher LOW-Byte
C00E STA $A9        nach Zeiger auf Programm LOW-Byte
C010 LDA $2C        BASIC-Anfang HIGH-Byte
C012 SBC #$00       minus Übertrag
C014 STA $AC        nach Zeiger für Zielspeicher HIGH-Byte
C016 STA $AA        nach Zeiger auf Programm HIGH-Byte
C018 LDY #$01       Programmzeiger auf Link-Zeiger setzen
C01A LDA ($A9),Y    Link-Byte LOW holen
C01C INY            Programmzeiger erhöhen
C01D ORA ($A9),Y    mit Link-Byte HIGH verknüpfen
C01F BEQ $C05E      verzweige, wenn Null
C021 INY            Programmzeiger erhöhen
C022 LDA ($A9),Y    Zeilenummer LOW laden
C024 TAX            nach X schieben
C025 INY            Programmzeiger erhöhen
C026 LDA ($A9),Y    Zeilenummer HIGH laden
C028 JSR $C213      prüfen, ob Zeilennummer in Tabelle
C02B BEQ $C079      verzweige, wenn ja
C02D LDY #$01       Programmzeiger auf Linker setzen
C02F LDA ($A9),Y    Linker LOW-Byte holen
C031 SEC            Carry für Subtraktion setzen
C032 SBC $A9        minus Programmzeiger LOW-Byte
C034 SEC            Carry für Subtraktion setzen
C035 SBC #$05       minus 5 ergibt gekürzte Zeilenlänge
C037 CLC            Carry für Addition löschen
C038 ADC $AD        plus bisherige Zeilenlänge
C03A BCS $C079      verzweige, wenn größer 255
C03C CMP #$F5       größer gleich $F5 (=245)?
C03E BCS $C079      verzweige, wenn ja
C040 STA $AD        Summe als neue Zeilenlänge speichern
C042 LDY #$00       Zähler für Verschiebeschleife auf Null
C044 LDA #$3A       ASCII ":"
```

```
C046 STA ($AB),Y    in Zielspeicher schreiben
C048 INY            Zeiger erhöhen
C049 LDA $A9        Programmzeiger LOW-Byte
C04B CLC            Carry für Addition löschen
C04C ADC #$04       plus 4
C04E STA $A9        nach Programmzeiger LOW-Byte
C050 BCC $C054      verzweige, wenn kein Übertrag
C052 INC $AA        HIGH-Byte Programmzeiger erhöhen
C054 LDA ($A9),Y    Programm-Byte holen
C056 BEQ $C090      verzweige, wenn nächste Zeile erreicht
C058 STA ($AB),Y    in Zielspeicher schreiben
C05A INY            Zähler erhöhen
C05B JMP $C054      Sprung zum Schleifenanfang
C05E TAY            Zeiger für Zielbereich auf Null
C05F STA ($AB),Y    Null ans Programmende schreiben
C061 INY            Zeiger erhöhen
C062 CPY #$03       schon drei Nullen geschrieben?
C064 BNE $C05F      verzweige, wenn nein
C066 TYA            Y-Register nach Akku
C067 CLC            Carry für Addition löschen
C068 ADC $AB        Programmende LOW-Byte berechnen
C06A STA $2D        in Programmendezeiger LOW speichern
C06C LDA $AC        Programmende HIGH-Byte
C06E ADC #$00       Übertrag addieren
C070 STA $2E        in Programmendezeiger HIGH speichern
C072 LDA #$37       BASIC-Interpreter
C074 STA $01        einschalten
C076 JMP $E1AB      CLR, Programmzeilen binden, Rücksprung
C079 LDY #$00       Zähler auf Null setzen
C07B LDA ($A9),Y    erste fünf Programm-Bytes
C07D STA ($AB),Y    verschieben
C07F INY            Zähler erhöhen
C080 CPY #$05       schon fünf Bytes verschoben?
C082 BNE $C07B      verzweige, wenn nein
C084 LDA ($A9),Y    Programm-Byte holen
C086 BEQ $C08E      verzweige, wenn nächste Zeile erreicht
C088 STA ($AB),Y    Programm-Byte speichern
C08A INY            Zähler erhöhen
C08B JMP $C084      Sprung zum Schleifenanfang
C08E STY $AD        Programmzeilenlänge speichern
```

```
C090 TYA          Zähler nach Akku
C091 CLC          Carry für Addition löschen
C092 ADC $A9      Programmzeiger LOW-Byte berechnen
C094 STA $A9      und speichern
C096 BCC $C09A    verzweige, wenn kein Übertrag
C098 INC $AA      Programmzeiger HIGH-Byte erhöhen
C09A TYA          Zähler nach Akku
C09B CLC          Carry für Addition löschen
C09C ADC $AB      Zielzeiger LOW-Byte berechnen
C09E STA $AB      und speichern
C0A0 BCC $C0A4    verzeige, wenn kein Übertrag
C0A2 INC $AC      Zielzeiger HIGH-Byte erhöhen
C0A4 JMP $C018    nächste Zeile bearbeiten
C0A7 LDA $2B      BASIC-Anfang LOW-Byte
C0A9 SEC          Carry für Subtraktion setzen
C0AA SBC #$01     minus 1
C0AC STA $AB      in Programmzeiger LOW-Byte speichern
C0AE LDA $2C      BASIC-Anfang HIGH-Byte
C0B0 SBC #$00     minus Übertrag
C0B2 STA $AC      in Programmzeiger HIGH-Byte speichern
C0B4 LDY #$00     LOW-Byte von $A000
C0B6 LDA #$A0     HIGH-Byte von $A000
C0B8 STY $A5      in Tabellenendezeiger LOW speichern
C0BA STA $A6      in Tabellenendezeiger HIGH speichern
C0BC LDY #$03     Programmzeiger auf erste Zeilennummer
C0BE LDA ($AB),Y  Zeilennummer LOW-Byte holen
C0C0 TAX          und nach X-Register schieben
C0C1 INY          Programmzeiger erhöhen
C0C2 LDA ($AB),Y  Zeilennummer HIGH-Byte holen
C0C4 JSR $C193    in Tabelle eintragen
C0C7 INY          Programmzeiger erhöhen
C0C8 LDA ($AB),Y  Programm-Byte holen
C0CA BNE $C0F6    verzweige, wenn nicht am Zeilenende
C0CC TYA          Zeiger nach Akku schieben
C0CD CLC          Carry für Addition löschen
C0CE ADC $AB      Programmzeiger berechnen
C0D0 STA $AB      und speichern
C0D2 BCC $C0D6    verzweige, wenn kein Übertrag
C0D4 INC $AC      Programmzeiger HIGH-Byte erhöhen
C0D6 LDY #$01     Zeiger auf Linker setzen
```

```
C0D8 LDA ($AB),Y   Linker LOW-Byte holen
C0DA INY           Programmzeiger erhöhen
C0DB ORA ($AB),Y   Akku mit Linker HIGH-Byte verknüpfen
C0DD BNE $C0E6     verzweige, wenn nicht am Programmende
C0DF STA $A7       Tabellenzeiger LOW auf Null setzen
C0E1 LDA #$A0      HIGH-Byte von $A000
C0E3 STA $A8       in Tabellenzeiger HIGH speichern
C0E5 RTS           Rücksprung
C0E6 LDA $02       Flag für IF-Befehl testen
C0E8 BEQ $C0F1     verzweige, wenn nicht gesetzt
C0EA LDA #$00      Flag für IF-Befehl
C0EC STA $02       löschen
C0EE JMP $C0BC     Zeile eintragen
C0F1 LDY #$05      Zeilennummer Überspringen
C0F3 JMP $C0C8     nächstes Programm-Byte testen
C0F6 CMP #$8D      GOSUB-Token?
C0F8 BEQ $C121     verzweige, wenn ja
C0FA CMP #$89      GOTO-Token?
C0FC BEQ $C121     verzweige, wenn ja
C0FE CMP #$CB      GO-Token?
C100 BEQ $C11A     verzweige, wenn ja
C102 CMP #$8B      IF-Token?
C104 BEQ $C17A     verzweige, wenn ja
C106 CMP #$A7      THEN-Token?
C108 BEQ $C121     verzweige, wenn ja
C10A CMP #$22      ASCII: Anführungszeichen?
C10C BEQ $C182     verzweige, wenn ja
C10E CMP #$8F      REM-Token?
C110 BEQ $C17A     verzweige, wenn ja
C112 CMP #$91      ON-Token?
C114 BEQ $C17A     verzweige, wenn ja
C116 INY           Programmzeiger erhöhen
C117 JMP $C0C8     nächstes Programm-Byte testen
C11A INY           Programmzeiger erhöhen
C11B LDA ($AB),Y   Programm-Byte holen
C11D CMP #$20      Leerzeichen?
C11F BEQ $C11A     verzweige, wenn ja
C121 INY           GOTO- bzw. TO-Token überspringen
C122 LDA ($AB),Y   Programm-Byte holen
C124 CMP #$20      Leerzeichen?
```

```
C126 BEQ $C121      verzweige, wenn ja
C128 LDA #$37       BASIC-Interpreter
C12A STA $01        einschalten
C12C STY $AE        Y-Register zwischenspeichern
C12E TYA            Zeiger in Akku schieben
C12F CLC            Carry für Addition löschen
C130 ADC $AB        Programmzeiger LOW-Byte berechnen
C132 STA $22        und speichern
C134 LDA $AC        Programmzeiger HIGH-Byte holen
C136 ADC #$00       Übertrag addieren
C138 STA $23        und speichern
C13A LDA ($AB),Y    Programm-Byte holen
C13C CMP #$30       kleiner als ASCII "0"?
C13E BCC $C148      verzweige, wenn ja
C140 CMP #$3A       größer als ASCII "9"?
C142 BCS $C148      verzweige, wenn ja
C144 INY            Zeiger erhöhen
C145 JMP $C13A      Sprung zum Schleifenanfang
C148 TYA            Y-Register auf
C149 PHA            Stapel retten
C14A SEC            Carry für Subtraktion setzen
C14B SBC $AE        Länge der Ziffernfolge berechnen
C14D BEQ $C171      verzweige, wenn gleich Null
C14F JSR $B7B5      Ziffern-String in Fließkommazahl wandeln
C152 JSR $B7F7      Fließkommazahl in Integerzahl wandeln
C155 PLA            Y-Register vom Stapel
C156 TAY            zurückholen
C157 LDA #$36       BASIC-Interpreter
C159 STA $01        wieder abschalten
C15B LDX $14        Zeilennummer LOW-Byte
C15D LDA $15        Zeilennummer HIGH-Byte
C15F JSR $C193      in Tabelle eintragen
C162 LDA ($AB),Y    nächtes Programm-Byte holen
C164 INY            Programmzeiger erhöhen
C165 CMP #$20       Leerzeichen?
C167 BEQ $C162      verzweige, wenn ja
C169 DEY            Programmzeiger verringern
C16A CMP #$2C       ASCII: "," (bei ON ... GOTO/GOSUB)?
C16C BEQ $C121      verzweige, wenn ja
C16E JMP $C0C8      nächstes Programm-Byte testen
```

```
C171 PLA          Y-Register vom Stapel
C172 TAY          zurückholen
C173 LDA #$36     BASIC-Interpreter
C175 STA $01      wieder abschalten
C177 JMP $C0C8    nächstes Programm-Byte testen
C17A LDA #$01     Flag für IF, REM oder ON
C17C STA $02      setzen
C17E INY          Token überspringen
C17F JMP $C0C8    nächstes Programm-Byte testen
C182 INY          Anführungszeichen überspringen
C183 LDA ($AB),Y  Programm-Byte holen
C185 BEQ $C18F    verzweige, wenn Zeile zu Ende
C187 CMP #$22     zweites Anführungszeichen erreicht?
C189 BNE $C182    verzweige, wenn nein
C18B INY          Programmzeiger erhöhen
C18C JMP $C0C8    nächstes Programm-Byte testen
C18F DEY          Programmzeiger verringern
C190 JMP $C17A    IF-Flag setzen
C193 STX $A9      Zeilennummer LOW
C195 STA $AA      Zeilennummer HIGH
C197 STY $AE      Y-Register zwischenspeichern
C199 LDY #$00     LOW-Byte von $A000
C19B STY $A7      nach Tabellenzeiger LOW schreiben
C19D LDA #$A0     HIGH-Byte von $A000
C19F STA $A8      nach Tabellenzeiger HIGH schreiben
C1A1 CPY $A5      Tab.-zeiger LOW mit Tab.-ende vergleichen
C1A3 LDA $A8      Tabellenzeiger HIGH
C1A5 SBC $A6      mit Tabellenende HIGH vergleichen
C1A7 BEQ $C1CD    verzweige, wenn Ende erreicht
C1A9 SEC          Carry für Subtraktion setzen
C1AA LDA ($A7),Y  Zeilennummer LOW aus Tabelle
C1AC SBC $A9      mit neuer Zeilennummer vergleichen
C1AE TAX          Differenz zwischenspeichern
C1AF INY          Tabellenzeiger erhöhen
C1B0 LDA ($A7),Y  Zeilennummer HIGH aus Tabelle
C1B2 SBC $AA      mit neuer Zeilennummer vergleichen
C1B4 INY          Programmzeiger LOW-Byte erhöhen
C1B5 BNE $C1B9    verzweige, wenn kein Übertrag
C1B7 INC $A8      Programmzeiger HIGH-Byte erhöhen
C1B9 BCC $C1A1    verzweige, wenn neue Zeilennr. größer
```

```
C1BB CMP #$00       HIGH-Byte der Zeilennummern gleich?
C1BD BNE $C1C5      verzweige, wenn nein
C1BF TXA            LOW-Byte der Zeilennummern gleich?
C1C0 BNE $C1C5      verzweige, wenn nein
C1C2 LDY $AE        Y-Register zurückholen
C1C4 RTS            Rücksprung
C1C5 DEY            Programmzeiger LOW-Byte verringern
C1C6 CPY #$FF       Übertrag?
C1C8 BNE $C1CC      verzweige, wenn nein
C1CA DEC $A8        Programmzeiger HIGH-Byte verringern
C1CC DEY            Programmzeiger LOW-Byte verringern
C1CD STY $A7        Y-Register in Programmz. LOW schreiben
C1CF LDA $A5        Tabellenende LOW
C1D1 STA $FB        in Verschiebezeiger LOW schreiben
C1D3 LDA $A6        Tabellenende HIGH
C1D5 STA $FC        in Verschiebezeiger HIGH schreiben
C1D7 LDA $A7        Tabellenzeiger LOW
C1D9 CMP $FB        mit Verschiebezeiger LOW vergleichen
C1DB LDA $A8        Tabellenzeiger HIGH
C1DD SBC $FC        mit Verschiebezeiger HIGH vergleichen
C1DF BCS $C1FD      verzweige, wenn Tab.-Zeiger größer
C1E1 DEC $FB        Verschiebezeiger LOW-Byte verringern
C1E3 LDA $FB        und holen
C1E5 CMP #$FF       Übertrag?
C1E7 BNE $C1EB      verzweige, wenn nein
C1E9 DEC $FC        Verschiebezeiger HIGH-Byte verringern
C1EB LDY #$00       Verschiebezeiger auf LOW-Byte vorher
C1ED LDA ($FB),Y    Zeilennummer LOW holen
C1EF LDY #$02       Verschiebezeiger auf LOW-Byte nachher
C1F1 STA ($FB),Y    Zeilennummer LOW speichern
C1F3 DEY            Verschiebezeiger auf HIGH-Byte vorher
C1F4 LDA ($FB),Y    Zeilennummer HIGH holen
C1F6 LDY #$03       Verschiebezeiger auf HIGH-Byte nachher
C1F8 STA ($FB),Y    Zeilennummer HIGH speichern
C1FA JMP $C1D7      Sprung zum Schleifenanfang
C1FD LDY #$00       Zeiger auf LOW-Byte in Tabelle setzen
C1FF LDA $A9        neue Zeilennummer LOW holen
C201 STA ($FB),Y    und in Tabelle schreiben
C203 INY            Zeiger auf HIGH-Byte in Tabelle setzen
C204 LDA $AA        neue Zeilennummer HIGH holen
```

```
C206 STA ($FB),Y    und in Tabelle schreiben
C208 INC $A5        Tabellenende LOW erhöhen
C20A INC $A5        Tabellenende LOW erhöhen
C20C BNE $C1C2      verzweige, wenn kein Übertrag
C20E INC $A6        Tabellenende HIGH erhöhen
C210 JMP $C1C2      Rücksprung
C213 STY $AE        Y-Register zwischenspeichern
C215 STX $FB        Zeilennummer LOW-Byte
C217 STA $FC        Zeilennummer HIGH-Byte
C219 LDA $A7        Tabellenzeiger LOW mit
C21B CMP $A5        Tabellenende LOW vergleichen
C21D LDA $A8        Tabellenzeiger HIGH mit
C21F SBC $A6        Tabellenende HIGH vergleichen
C221 BCS $C23D      verzweige, wenn Ende erreicht
C223 LDY #$00       Zeiger auf Null setzen
C225 LDA $FB        Zeilennummer LOW holen
C227 CMP ($A7),Y    mit Tabellenwert vergleichen
C229 INY            Zeiger erhöhen
C22A LDA $FC        Zeilennummmer HIGH holen
C22C SBC ($A7),Y    mit Tabellenwert vergleichen
C22E BCC $C23D      verzweige, wenn nicht in Tabelle
C230 INC $A7        Tabellenzeiger LOW erhöhen
C232 INC $A7        Tabellenzeiger LOW erhöhen
C234 BNE $C238      verzweige, wenn kein Übertrag
C236 INC $A8        Tabellenzeiger HIGH erhöhen
C238 LDY $AE        Y-Register zurückholen
C23A LDA #$00       Zero-Flag setzen
C23C RTS            Rücksprung
C23D LDY $AE        Y-Register zurückholen
C23F LDA #$01       Zero-Flag löschen
C241 RTS            Rücksprung
```

Listschutz:

```
C242 LDY #$00       Y-Register auf Null setzen
C244 LDA $2D        BASIC-Programmende LOW-Byte
C246 STA $AB        nach Verschiebezeiger 1 LOW-Byte
C248 LDA $2E        BASIC-Programmende HIGH-Byte
C24A STA $AC        nach Verschiebezeiger 1 HIGH-Byte
```

```
C24C LDA $37       BASIC-Speicherende LOW-Byte
C24E STA $A9       nach Verschiebezeiger 2 LOW-Byte
C250 LDA $38       BASIC-Speicherende HIGH-Byte
C252 STA $AA       nach Verschiebezeiger 2 HIGH-Byte
C254 DEC $AB       Verschiebezeiger 1 LOW verringern
C256 LDA $AB       und holen
C258 CMP #$FF      Übertrag?
C25A BNE $C25E     verzweige, wenn nein
C25C DEC $AC       Verschiebezeiger 1 HIGH verringern
C25E DEC $A9       Verschiebezeiger 2 LOW verringern
C260 LDA $A9       und holen
C262 CMP #$FF      Übertrag?
C264 BNE $C268     verzweige, wenn nein
C266 DEC $AA       Verschiebezeiger 2 HIGH verringern
C268 LDA ($AB),Y   Programm-Byte holen und
C26A STA ($A9),Y   ans Speicherende verschieben
C26C LDA $AB       Verschiebezeiger 1 LOW
C26E CMP $2B       mit BASIC-Anfang LOW vergleichen
C270 LDA $AC       Verschiebezeiger 1 HIGH
C272 SBC $2C       mit BASIC-Anfang HIGH vergleichen
C274 BCS $C254     verzweige, wenn Anfang nicht erreicht
C276 LDY #$01      Programmzeiger auf Linker setzen
C278 LDA ($A9),Y   Linker LOW-Byte holen
C27A INY           Programmzeiger erhöhen
C27B ORA ($A9),Y   mit Linker HIGH-Byte verknüpfen
C27D BEQ $C2C2     verzweige, wenn Programmende erreicht
C27F LDY #$00      Programmzeiger auf Zeilenanfang
C281 LDA ($A9),Y   Zeilenanfang holen
C283 STA ($AB),Y   und in Zielbereich verschieben
C285 INY           Programmzeiger erhöhen
C286 CPY #$05      schon fünf Bytes verschoben?
C288 BNE $C281     verzweige, wenn nein
C28A LDA $AB       Zielzeiger LOW holen
C28C CLC           Carry für Addition löschen
C28D ADC #$05      plus 5
C28F STA $AB       in Zielzeiger LOW schreiben
C291 BCC $C295     verzweige, wenn kein Übertrag
C293 INC $AC       Zielzeiger HIGH erhöhen
C295 LDY #$00      Zielzeiger-Index um fünf verringern
C297 TYA           Akku gleich Null setzen
```

```
C298 STA ($AB),Y    und in Zielbereich schreiben
C29A INY            Zielzeiger erhöhen
C29B LDA #$3A       ASCII: ":"
C29D CPY #$05       schon fünf Doppelpunkte eingefügt?
C29F BNE $C298      verzweige, wenn nein
C2A1 LDA ($A9),Y    Programm-Byte holen
C2A3 BEQ $C2AB      verzweige, wenn Zeilenende erreicht
C2A5 STA ($AB),Y    Byte in Zielbereich schreiben
C2A7 INY            Zeiger erhöhen
C2A8 JMP $C2A1      nächstes Programm-Byte verschieben
C2AB TYA            Y-Register in Akku schieben
C2AC CLC            Carry für Addition löschen
C2AD ADC $A9        Programmzeiger LOW berechnen
C2AF STA $A9        und speichern
C2B1 BCC $C2B5      verzweige, wenn kein Übertrag
C2B3 INC $AA        Programmzeiger HIGH erhöhen
C2B5 TYA            Y-Register in Akku schieben
C2B6 CLC            Carry für Addition löschen
C2B7 ADC $AB        Zielzeiger LOW berechnen
C2B9 STA $AB        und speichern
C2BB BCC $C276      verzweige, wenn kein Übertrag
C2BD INC $AC        Zielzeiger HIGH erhöhen
C2BF JMP $C276      nächste Zeile bearbeiten
C2C2 LDY #$00       Zeiger auf Null setzen
C2C4 TYA            Null in Akku laden
C2C5 STA ($AB),Y    und an Programmende schreiben
C2C7 INY            Zeiger erhöhen
C2C8 CPY #$03       schon drei Nullen geschrieben?
C2CA BNE $C2C5      verzweige, wenn nein
C2CC TYA            Y-Register in Akku schieben
C2CD CLC            Carry für Addition löschen
C2CE ADC $AB        Programmende LOW berechnen und in
C2D0 STA $2D        Programmendezeiger LOW speichern
C2D2 LDA $AC        Programmende HIGH holen
C2D4 ADC #$00       Übertrag addieren und in
C2D6 STA $2E        Programmendezeiger HIGH speichern
C2D8 JMP $E1AB      CLR, Programmzeilen binden, Rücksprung
```

Nachfolgend der BASIC-Loader:

```
100 FORI=1TO731STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATAA9,36,85,01,20,A7,C0,A5,2B,38,E9,01,85,AB,85, 147
301 DATAA9,A5,2C,E9,00,85,AC,85,AA,A0,01,B1,A9,C8,11, 151
302 DATAA9,F0,3D,C8,B1,A9,AA,C8,B1,A9,20,13,C2,F0,4C, 245
303 DATAA0,01,B1,A9,38,E5,A9,38,E9,05,18,65,AD,B0,3D, 254
304 DATAC9,F5,B0,39,85,AD,A0,00,A9,3A,91,AB,C8,A5,A9, 174
305 DATA18,69,04,85,A9,90,02,E6,AA,B1,A9,F0,38,91,AB, 147
306 DATAC8,4C,54,C0,A8,91,AB,C8,C0,03,D0,F9,98,18,65, 117
307 DATAAB,85,2D,A5,AC,69,00,85,2E,A9,37,85,01,4C,AB, 39
308 DATAE1,A0,00,B1,A9,91,AB,C8,C0,05,D0,F7,B1,A9,F0, 181
309 DATA06,91,AB,C8,4C,84,C0,84,AD,98,18,65,A9,85,A9, 183
310 DATA90,02,E6,AA,98,18,65,AB,85,AB,90,02,E6,AC,4C, 130
311 DATA18,C0,A5,2B,38,E9,01,85,AB,A5,2C,E9,00,85,AC, 229
312 DATAA0,00,A9,A0,84,A5,85,A6,A0,03,B1,AB,AA,C8,B1, 95
313 DATAAB,20,93,C1,C8,B1,AB,D0,2A,98,18,65,AB,85,AB, 45
314 DATA90,02,E6,AC,A0,01,B1,AB,C8,11,AB,D0,07,85,A7, 168
315 DATAA9,A0,85,A8,60,A5,02,F0,07,A9,00,85,02,4C,BC, 172
316 DATAC0,A0,05,4C,C8,C0,C9,8D,F0,27,C9,89,F0,23,C9, 212
317 DATACB,F0,18,C9,8B,F0,74,C9,A7,F0,17,C9,22,F0,74, 81
318 DATAC9,8F,F0,68,C9,91,F0,64,C8,4C,C8,C0,C8,B1,AB, 30
319 DATAC9,20,F0,F9,C8,B1,AB,C9,20,F0,F9,A9,37,85,01, 46
320 DATA84,AE,98,18,65,AB,85,22,A5,AC,69,00,85,23,B1, 172
321 DATAAB,C9,30,90,08,C9,3A,B0,04,C8,4C,3A,C1,98,48, 226
322 DATA38,E5,AE,F0,22,20,B5,B7,20,F7,B7,68,A8,A9,36, 38
323 DATA85,01,A6,14,A5,15,20,93,C1,B1,AB,C8,C9,20,F0, 107
324 DATAF9,88,C9,2C,F0,B3,4C,C8,C0,68,A8,A9,36,85,01, 98
325 DATA4C,C8,C0,A9,01,85,02,C8,4C,C8,C0,C8,B1,AB,F0, 181
326 DATA08,C9,22,D0,F7,C8,4C,C8,C0,88,4C,7A,C1,86,A9, 148
327 DATA85,AA,84,AE,A0,00,84,A7,A9,A0,85,A8,C4,A5,A5, 176
328 DATAA8,E5,A6,F0,24,38,B1,A7,E5,A9,AA,C8,B1,A7,E5, 20
329 DATAAA,C8,D0,02,E6,A8,90,E6,C9,00,D0,06,8A,D0,03, 68
330 DATAA4,AE,60,88,C0,FF,D0,02,C6,A8,88,84,A7,A5,A5, 54
331 DATA85,FB,A5,A6,85,FC,A5,A7,C5,FB,A5,A8,E5,FC,B0, 54
```

```
332 DATA1C,C6,FB,A5,FB,C9,FF,D0,02,C6,FC,A0,00,B1,FB, 37
333 DATAA0,02,91,FB,88,B1,FB,A0,03,91,FB,4C,D7,C1,A0, 21
334 DATA00,A5,A9,91,FB,C8,A5,AA,91,FB,E6,A5,E6,A5,D0, 99
335 DATAB4,E6,A6,4C,C2,C1,84,AE,86,FB,85,FC,A5,A7,C5, 84
336 DATAA5,A5,A8,E5,A6,B0,1A,A0,00,A5,FB,D1,A7,C8,A5, 108
337 DATAFC,F1,A7,90,0D,E6,A7,E6,A7,D0,02,E6,A8,A4,AE, 253
338 DATAA9,00,60,A4,AE,A9,01,60,A0,00,A5,2D,85,AB,A5, 172
339 DATA2E,85,AC,A5,37,85,A9,A5,38,85,AA,C6,AB,A5,AB, 54
340 DATAC9,FF,D0,02,C6,AC,C6,A9,A5,A9,C9,FF,D0,02,C6, 41
341 DATAAA,B1,AB,91,A9,A5,AB,C5,2B,A5,AC,E5,2C,B0,DE, 112
342 DATAA0,01,B1,A9,C8,11,A9,F0,43,A0,00,B1,A9,91,AB, 230
343 DATAC8,C0,05,D0,F7,A5,AB,18,69,05,85,AB,90,02,E6, 210
344 DATAAC,A0,00,98,91,AB,C8,A9,3A,C0,05,D0,F7,B1,A9, 177
345 DATAF0,06,91,AB,C8,4C,A1,C2,98,18,65,A9,85,A9,90, 37
346 DATA02,E6,AA,98,18,65,AB,85,AB,90,B9,E6,AC,4C,76, 31
347 DATAC2,A0,00,98,91,AB,C8,C0,03,D0,F9,98,18,65,AB, 74
348 DATA85,2D,A5,AC,69,00,85,2E,4C,AB,E1,A4,A4,A4,A4, 135
```

2.1.5 Simuliertes Programmende

Einen perfekten LIST- und RUN-Schutz gleichzeitig erhält man, wenn man dem BASIC-Interpreter "vorgaukelt", daß das im Speicher befindliche Programm zu Ende sei, obwohl es noch gar nicht zu Ende ist. Der BASIC-Interpreter erkennt anhand seiner Markierung (drei hintereinander folgende Nullen), ab welcher Speicheradresse das BASIC-Programm zu Ende ist.

Wird nun ein BASIC-Programm gelistet oder gerade abgearbeitet, prüft der BASIC-Interpreter nach jedem Befehl, ob das BASIC-Programm zu Ende ist. Hinter einem BASIC-Programm stehen immer drei Nullen als Endmarkierung. Trifft nun der BASIC-Interpreter beim Listen oder beim Abarbeiten eines BASIC-Programms auf diese Markierung, unterbricht er automatisch den jeweiligen Vorgang.

Wir möchten Ihnen anhand dieses kleinen BASIC-Programms den Schutz demonstrieren:

```
10 PRINT "HALLO"
20 REM
30 GOTO 10
```

Unser LIST-Schutz basiert nun auf dieser Tatsache. Es ist durchaus möglich, diese Markierung mitten in ein BASIC-Programm zu setzen, ohne daß dieses zerstört wird. Und zwar ist das nur am Ende einer BASIC-Zeile möglich, weil dort eine Markierung für das Zeilenende steht. Die Markierung für Zeilenende besteht nur aus einer Null. Hinter dieser Null folgt nun der Linker,der auf die nächste Zeile des BASIC-Programms zeigt.

Im Maschinensprachemonitor sieht das BASIC-Programm folgendermaßen aus:

```
0800: 00 0F 08 0A 00 99 20 22   ..... "
0808: 48 41 4C 4C 4F 22 00 15   HALLO"..
0810: 08 14 00 8F 00 1E 08 1E   ........
0818: 00 89 20 31 30 00 00 00   ..10....
```

Wenn man sich nun den Wert dieser beiden Speicherzellen, die den Linker darstellen, auf ein Blatt Papier aufschreibt und sie anschließend mit dem Wert Null überschreibt, haben wir mit der Zeilenendmarkierung drei Nullen mitten im BASIC-Programm stehen. Wichtig ist auch, daß man sich die Adresse des geänderten Linkers aufschreibt, um später die richtigen Werte per POKE wieder hineinzuschreiben. Der Anfang der nächsten Zeile ist daran zu erkennen, daß hinter der Null vier weitere Bytes folgen. Die ersten beiden stellen, wie schon erwähnt, den Linker dar und die anderen beiden Bytes die Zeilennummer.

Hier die BASIC-Zeile ohne LINKER:

```
0800: 00 0F 08 0A 00 99 20 22   ..... "
0808: 48 41 4C 4C 4F 22 00 00   HALLO"..
0810: 00 14 00 8F 00 1E 08 1E   ........
0818: 00 89 20 31 30 00 00 00   ..10....
```

Wenn man nun das veränderte BASIC-Programm listen möchte, sieht man es nur bis zu der Adresse, an der der BASIC-Interpreter auf diese Nullen trifft. Dort wird dann der LIST-Vorgang automatisch unterbrochen. Genauso verhält es sich, wenn man versucht, das Programm zu starten. Es wird nur bis zu der Stelle laufen, an der die Nullen beginnen, und nicht weiter.

Bei unserem BASIC-Programm sieht das folgendermaßen aus:

```
LIST
10 PRINT "HALLO"

READY.

RUN
HALLO

READY.
```

Das Programm läßt sich auch in diesem Zustand auf Diskette speichern. Wenn man es wieder in den Speicher lädt, zeigt es die gleichen Symptome wie vorher. Um nun diese wieder zu entfernen (falls man sein Programm wieder starten möchte), muß man die Originalwerte, die wir vorher gelöscht haben, wieder ins BASIC-Programm hineinschreiben. Dazu brauchen wir einfach nur zwei POKEs im Direktmodus einzugeben. Poken Sie nur die Adresse des Linkers und die Adresse des Linkers+1 mit den entsprechenden Werten in den Speicher. Nun läßt sich das Programm wieder listen und auch starten.

Noch ein wichtiger Hinweis: Im geschützten Zustand des BASIC-Programms darf man auf keinen Fall Änderungen am BASIC-Programm vornehmen, da sonst die Adressen verschoben

werden und die Adresse des Linkers nicht mehr stimmt. Beim Zurückstellen könnten Sie dann das BASIC-Programm zerstören.

2.1.6 Der Linke(r)-Trick

Man erhält einen hervorragenden LIST-Schutz, wenn man den ersten LINKER des BASIC-Programms in eine Endlosschleife zeigen läßt. Man muß nur den LINKER wieder auf $0801 zeigen lassen.

Das kleine BASIC-Programm vorher:

```
10 PRINT "HALLO"
20 REM
30 GOTO 10

0800: 00 0F 08 0A 00 99 20 22   ..... "
0808: 48 41 4C 4C 4F 22 00 15   HALLO"..
0810: 08 14 00 8F 00 1E 08 1E   ........
0818: 00 89 20 31 30 00 00 00   ..10....
```

Jetzt braucht man nur in die Speicherzellen $0801 und $0802 die Werte $01 und $08 hineinzuschreiben. Damit haben Sie den LINKER in eine Endlosschleife zeigen lassen.

Wenn man nun LIST eingibt, sieht dieses folgendermaßen aus:

```
10 PRINT "HALLO"
10 PRINT "HALLO"
10 PRINT "HALLO"
10 PRINT "HALLO"
10 PRINT "HALLO"
10 PRINT "HALLO"
10 PRINT "HALLO"
.
.
.
```

Sie werden feststellen, daß der BASIC-Interpreter immer nur ein und dieselbe Zeile listet und nicht zu den anderen kommt. Bei diesem Schutz kann das Programm normal mit RUN gestartet werden.

Es ist auch nicht möglich, die dahinterliegenden Zeilen zu löschen. Wenn man aber die erste Zeile löscht, normalisiert sich das Ganze wieder von selbst.

Von daher ist es besser, mit dieser Methode Programmteile zu überbrücken, das heißt, den LINKER so zu stellen, daß er ein paar BASIC-Zeilen überspringt und dann erst weiter listet.

In unserem Programmbeispiel könnte das so aussehen:

```
10 PRINT "HALLO"
30 GOTO 10
```

Hier haben wir nur den LINKER auf die übernächste Zeile verzweigen lassen. Die REM-Zeile ist aber dennoch vorhanden, sie wird nur nicht gelistet. Beim RUN würde diese Zeile, falls dort ein anderer Befehl stehen würde, trotzdem ausgeführt werden.

Diese Methode ist vollkommen unauffällig. Das hat den Vorteil, daß man wichtige BASIC-Zeilen, in denen zum Beispiel ein Password abgefragt oder überprüft wird, einfach überspringt und damit unsichtbar macht.

Noch ein wichtiger Hinweis: Die Zeilen, die man überspringt, also unsichtbar macht, dürfen keinesfalls von einem GOTO, GOSUB oder THEN-Befehl angesprungen werden, da der BASIC-Interpreter die Zeile nicht finden kann und man die Fehlermeldung "UNDEF'D STATEMENT ERROR" erhält. Das BASIC-Programm würde dann nicht einwandfrei laufen.

2.1.7 Der Sys-Bluff

Hier möchten wir Ihnen noch eine sehr wirkungsvolle LIST-Schutz-Methode demonstrieren: es läßt sich ein hervorragender LIST-Schutz erreichen, indem man den BASIC-Start verschiebt. Aber dazu später. Der LIST-Schutz soll wie folgt aussehen:

```
10 sys 2064
```

Beim Listen erscheint nur eine SYS-Zeile, so daß der Fremdbenutzer denken muß, es handle sich hier um ein Maschinenprogramm. Wir haben Ihnen hier ein Schutzprogramm erstellt, das den Einbau der SYS-Zeile selbst vornimmt.

Das Maschinenprogramm wird mit SYS 49152 ($C000) gestartet. Das zu schützende BASIC-Programm muß sich auf Diskette befinden. Es muß sich um ein reines BASIC-Programm handeln, das keine Maschinenprogramm-Unterroutinen besitzt, weil die Gefahr besteht, daß diese nicht mehr aufgerufen werden können. Das Schutzprogramm fragt nach dem Start nach dem Filenamen des zu schützenden BASIC-Programms. Anschließend wird dieses geladen, mit dem Schutz versehen und direkt wieder auf Diskette gespeichert. Der alte Filename wird beibehalten. Es wird nur ein Zusatz dahintergehängt, damit man erkennt, daß es sich um das geschützte Programm handelt. Deshalb sollte der Filename des zu schützenden BASIC-Programms nicht länger als 14 Zeichen sein.

Nun zur Funktionsweise:

Nach dem Eingeben des Filenamens kopiert das Schutzprogramm die SYS-Zeile und das dazugehörige Maschinenprogramm in den BASIC-Speicher ab der Adresse $0801 (2049). Anschließend wird das zu schützende BASIC-Programm direkt dahinter geladen. Beide Teile, also die SYS-Zeile und das BASIC-Programm, werden wieder auf Diskette gespeichert.

Beim Listen sieht man nur noch die SYS-Zeile. Nach einem RUN führt die SYS-Zeile das dahinterliegende Maschinenpro-

gramm aus, das die BASIC-Zeiger auf das dahinterliegende BASIC-Programm setzt. Vor dem Programmstart werden noch die BASIC-Linker neu gebunden, um diese an den neuen Speicherbereich anzupassen.

Nun das versprochene Schutzprogramm in Maschinencode und BASIC-Loader:

```
C000 LDX #$00     Zähler auf Null setzen
C002 LDA $C080,X  Erstes Byte der BASIC-Zeile holen
C005 STA $0801,X  und in BASIC-Speicher schreiben
C008 INX          Zähler erhöhen
C009 CPX #$28     Falls ganze BASIC-Zeile geschrieben,
C00B BNE $C002    dann weiter
C00D LDA #$93     Bildschirm
C00F JSR $FFD2    löschen
C012 LDX #$00     PROGRAMMNAME
C014 LDA $C0A8,X  auf den
C017 JSR $FFD2    Bildschirm
C01A INX          bringen
C01B CPX #$0E     Falls alle Zeichen,
C01D BNE $C014    dann weiter
C01F LDX #$00     Zähler auf Null
C021 JSR $FFCF    Zeichen holen
C024 STA $C0B8,X  und speichern
C027 INX          Zähler erhöhen
C028 CMP #$0D     Falls RETURN gedrückt weiter
C02A BNE $C021    ansonsten nächstes Zeichen holen
C02C DEX          Byte-Anzahl des Filenamen
C02D STX $02      speichern
C02F LDX #$08     Geräteadresse
C031 LDY #$00     Sekundäradresse
C033 JSR $FFBA    File-Parameter setzen
C036 LDA $02      Byte-Anzahl des Filenamen holen
C038 LDX #$B8     Anfangsadresse des
C03A LDY #$C0     Filenamen setzen
C03C JSR $FFBD    Filenamenparameter setzen
C03F LDA #$00     Zeichen für LOAD setzen
C041 LDX #$29     LOW-Byte Anfangsadresse
```

```
C043 LDY #$08     HIGH-BYTE Anfangsadresse
C045 JSR $FFD5    LOAD-Befehl
C048 LDX $02      Byte-Anzahl des Filenamen holen
C04A LDA #$2F     Code für "/" laden
C04C STA $C0B8,X und hinter Filenamen speichern
C04F LDA #$53     Code für "S" laden
C051 STA $C0B9,X und hinter Filenamen speichern
C054 INX          Byte-Anzahl des Filenamen um
C055 INX          zwei erhöhen
C056 STX $02      und speichern
C058 LDX #$08     Geräteadresse
C05A LDY #$01     Sekundäradresse
C05C JSR $FFBA    File-Parameter setzen
C05F LDA $02      Byte-Anzahl des Filenamens laden
C061 LDX #$B8     Anfangsadresse des
C063 LDY #$C0     Filenamens laden
C065 JSR $FFBD    Filenamenparameter setzen
C068 LDX #$01     Anfangsadresse
C06A LDY #$08     setzen
C06C STX $FB      und
C06E STY $FC      speichern
C070 LDA #$FB     Zeiger auf Anfangsadresse setzen
C072 LDX $AE      Endadresse
C074 LDY $AF      holen
C076 JSR $FFD8    SAVE-Befehl
C079 RTS          Rücksprung

C080 0C 08 0A 00 9E 20 32 30 BASIC-Zeile
C088 36 34 00 00 00 00 00 18
```

Maschinenprogramm hinter der SYS-Zeile

```
C08F CLC          Carry-Flag löschen
C090 LDA #$28     Zeichen-Anzahl mit
C092 ADC $2B      BASIC-Start-LOW-Byte addieren
C094 STA $2B      und speichern
C096 LDA #$00     Null laden, falls Übertrag: 1 laden
C098 ADC $2C      BASIC-Start-HIGH-Byte addieren
C09A STA $2C      und speichern
```

```
C09C JSR $A659   CHRGET-Zeiger auf Programmstart und CLR
C09F JSR $A533   BASIC-LINKER anpassen
C0A2 JMP $A7AE   in die Interpreterschleife springen
C0A5 BRK
C0A6 BRK
C0A7 BRK

C0A8 50 52 4F 47 52 41 4D 4D PROGRAMM
C0B0 4E 41 4D 45 3A 20 00 00 NAME: ..
```

Nachfolgend der dazugehörige BASIC-Loader:

```
100 FORI=1TO184STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A:NEXT:READA:
IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA A2,00,BD,80,C0,9D,01,08,E8,E0,28,D0,F5,A9,93, 54
301 DATA 20,D2,FF,A2,00,BD,A8,C0,20,D2,FF,E8,E0,0E,D0, 79
302 DATA F5,A2,00,20,CF,FF,9D,B8,C0,E8,C9,0D,D0,F5,CA, 231
303 DATA 86,02,A2,08,A0,00,20,BA,FF,A5,02,A2,B8,A0,C0, 12
304 DATA 20,BD,FF,A9,00,A2,29,A0,08,20,D5,FF,A6,02,A9, 61
305 DATA 2F,9D,B8,C0,A9,53,9D,B9,C0,E8,E8,86,02,A2,08, 88
306 DATA A0,01,20,BA,FF,A5,02,A2,B8,A0,C0,20,BD,FF,A2, 89
307 DATA 01,A0,08,86,FB,84,FC,A9,FB,A6,AE,A4,AF,20,D8, 237
308 DATA FF,60,00,00,00,00,00,00,0C,08,0A,00,9E,20,32, 109
309 DATA 30,36,34,00,00,00,00,00,18,A9,28,65,2B,85,2B, 195
310 DATA A9,00,65,2C,85,2C,20,59,A6,20,33,A5,4C,AE,A7, 163
311 DATA 00,00,00,50,52,4F,47,52,41,4D,4D,4E,41,4D,45, 134
312 DATA 3A,20,00,00,48,41,4C,4C,4F,2F,53,00,00,00,00, 76
```

Es ist auch vorteilhaft, vor dem Schützen seines BASIC-Programms in der ersten Zeile mit POKE 808,225 die RUN-STOP-RESTORE-Taste außer Betrieb zu setzen, um ein späteres Listen zu vermeiden. Zusätzlich kann im BASIC-Programm selbst noch ein zusätzlicher Listschutz eingebaut werden, um sicher zu gehen, daß wirklich niemand das Programm listen kann.

2.2 Änderungsschutz

2.2.1 Abfrage des BASIC-Endes

Eine einfache, aber wirkungsvolle Methode, das Ändern eines BASIC-Programms zu verhindern, ist folgende: Man kann innerhalb eines BASIC-Programms das BASIC-Programmende mittels PEEK ermitteln und es nachher im BASIC-Programm abfragen. Das Ende eines BASIC-Programms ermittelt man folgendermaßen:

```
PRINT PEEK(45)+PEEK(46)*256
```

Da das BASIC-Programmende, das in den Speicherzellen $2D (45) und $2E (46) in LOW- und HIGH-Byte Darstellung verzeichnet ist, muß man, wie oben gezeigt, in eine ganze Zahl umrechnen, damit man es später im BASIC-Programm abfragen kann.

Bevor man sich nun das BASIC-Programmende holt und berechnet, muß man die Programmzeile, in der die Abfrage stattfinden soll, vorher in das zu schützende BASIC-Programm einbauen, weil sich beim Einfügen einer neuen BASIC-Zeile das BASIC-Programmende wieder verschieben würde. Dies hätte zur Folge, daß die berechnete Prüfsumme nicht mehr stimmen würde.

Die Abfrage könnte so aussehen:

```
10 A=PEEK(45)+PEEK(46)*256
20 IF A<>00000 THEN POKE 776,1
```

Wenn man diese Zeilen in das zu schützende BASIC-Programm einbaut und dann im Direktmodus das BASIC-Programmende berechnet, erhält man den korrekten Wert, den man in Zeile 20 eintragen kann. Man darf auf keinen Fall beim Ändern der Prüfsumme eine Zahl löschen oder hinzufügen, da sich die Prüfsumme wieder verändern würde.

Wenn die Prüfsumme zum Beispiel 2543 betragen würde, müßte man diese in Zeile 20 folgendermaßen eintragen:

```
20 IF A<>02543 THEN POKE 776,1
```

Es ist vorteilhafter, wenn man diese Abfrage irgendwo mitten im BASIC-Programm einbaut, um sie besser zu verbergen. Am besten versieht man das BASIC-Programm anschließend noch mit einem guten LIST-Schutz.

Falls nun eine Änderung an diesem BASIC-Programm erfolgt, das heißt, eine Zeile bzw. ein Zeichen eingefügt oder gelöscht wird, verschiebt sich das BASIC-Programmende automatisch. Dadurch würde der Wert in Zeile 20 nicht mehr mit dem BASIC-Programmende übereinstimmen, was zur Folge hätte, daß der dahinterliegende POKE-Befehl ausgeführt würde.

Dieser POKE-Befehl würde den Vektor für "BASIC-Befehlsadresse holen" umändern, so daß dann kein BASIC-Befehl ausgeführt werden kann. Das BASIC-Programm könnte also weder gestartet, gelistet noch per NEW gelöscht werden. Dies hätte zur Folge, daß man einen Systemreset ausführen müßte.

2.2.2 Änderungsschutz durch übergroße Zeilennummer

Wir haben hier für Sie ein kleines Programm vorbereitet, das einen sehr guten Änderungsschutz in Ihr BASIC-Programm einbaut. Das Programm funktioniert folgendermaßen:

Das zu schützende Programm darf keine Zeilennummern 1 und 2 enthalten, weil diese vom Schutz-Programm generiert werden. Das Schutzprogramm liegt sowohl im Maschinencode als auch als BASIC-Loader vor. Nach dem Start steht das Maschinenprogramm ab der Adresse $C000 (49152) zur Verfügung. Es wird vom BASIC-Loader direkt mit SYS 49152 gestartet. Nach dem Start fragt das Programm nach dem Filenamen des zu schützenden BASIC-Programms. Nach der Eingabe des Filenamens wird

das BASIC-Programm von Diskette geladen und in geschützter Form wieder auf Diskette gespeichert. Das mit dem Änderungsschutz versehene Programm ist um nur wenige Bytes verlängert.

Nun zur Funktionsweise:

Das Maschinenprogramm generiert zwei BASIC-Zeilen mit den Zeilennummern 1 und 2. Die Zeile 1 ist eine REM-Zeile, hinter der noch zwei kurze Maschinenprogramme stehen, deren Funktion im folgenden noch erläutert wird. In der zweiten Zeile steht ein SYS-Befehl, der eines der beiden Maschinenprogramme in Zeile 0 startet.

Sind diese beiden Zeilen nun erzeugt, wird die Zeilennummer 0 durch eine höhere, eigentlich unerlaubte Zeilennummer (größer als 64000) ersetzt. Diese Zeile kann daher auch nicht gelöscht werden.

Da alle nun folgenden Zeilen kleiner sind als die erste, können diese vom Computer nicht mehr erkannt werden. Ein Sprung in eine solche Zeile führt zu der Fehlermeldung: "UNDEF'D STATEMENT ERROR". Es kann daher keine Zeile gelöscht werden, da diese für den Interpreter nicht mehr vorhanden ist.

Der einzige Nachteil ist, daß es nicht nur ein perfekter Löschschutz, sondern auch ein RUN-Schutz ist, weil Sprungziele innerhalb des Programms nicht mehr gefunden werden können.

Wird das BASIC-Programm nun gestartet, trifft der Interpreter zuerst auf den SYS-Befehl in Zeile 2. Es folgt ein Sprung in das Maschinenprogramm in der REM-Zeile. Dort wird die Zeilennummer wieder auf 1 gesetzt, und der Vektor auf EINGABE EINER ZEILE wird auf die zweite Maschinenroutine gesetzt.

Nun kann das BASIC-Programm ohne Fehler ausgeführt werden. Wird der Programmablauf zum Beispiel durch die STOP-Taste, Fehlermeldungen, Programmende und so weiter unterbrochen, wird das zweite Maschinenprogramm über den geänderten Vektor angesprungen. Dort wird die Zeilennummer wieder erhöht,

der Vektor für "Eingabe einer Zeile" wieder auf den normalen Wert gebracht und die Routine für "Eingabe einer Zeile" angesprungen. Das Programm liegt nun wieder in der geschützten Form vor.

Hier nun das Schutz-Programm:

```
C000 LDX #$00     Zähler auf Null stellen
C002 LDA $C080,X BASIC-Zeile ins
C005 STA $0801,X BASIC-RAM kopieren
C008 INX          Zähler erhöhen
C009 CPX #$3B     Falls alle Zeichen, dann weiter
C00B BNE $C002    ansonsten nächstes Zeichen holen
C00D JSR $E544    Bildschirm löschen
C010 LDX #$00     Zähler auf Null
C012 LDA $C0C0,X PROGRAMMNAME auf
C015 JSR $FFD2    Bildschirm bringen
C018 INX          Zähler erhöhen
C019 CPX #$0E     Falls alle Zeichen, dann weiter
C01B BNE $C012    ansonsten nächstes Zeichen holen
C01D LDX #$00     Zähler auf Null stellen
C01F JSR $FFCF    Byte von Tastatur holen
C022 STA $C0D0,X und speichern
C025 INX          Zähler erhöhen
C026 CMP #$0D     Falls noch kein RETURN, dann nächstes
C028 BNE $C01F    Zeichen holen
C02A DEX          Zähler erniedrigen
C02B STX $02      und speichern
C02D LDX #$08     Geräteadresse laden
C02F LDY #$00     Sekundäradresse laden
C031 JSR $FFBA    File-Parameter setzen
C034 LDA $02      Byte-Anzahl des Filennamens laden
C036 LDX #$D0     Position des
C038 LDY #$C0     Filenamens laden
C03A JSR $FFBD    Filenamenparameter setzen
C03D LDA #$00     Flag für LOAD setzen
C03F LDX #$3C     Startadresse
C041 LDY #$08     festlegen
C043 JSR $FFD5    LOAD-Befehl
```

```
C046 LDA $AE      LOW- und
C048 STA $2D      HIGH-Byte
C04A LDA $AF      für BASIC-.,C04C STA $2E     Ende setzen
C04E LDX $02      Byte-Anzahl des Filenamens laden
C050 LDA #$2F     Code für "/" laden
C052 STA $C0D0,X  und hinter Filenamen speichern
C055 INX          Zähler erhöhen
C056 LDA #$41     Code für "A" laden
C058 STA $C0D0,X  und hinter Filenamen speichern
C05B INX          Zähler wieder erhöhen
C05C STX $02      und speichern
C05E LDX #$08     Geräteadresse laden
C060 LDY #$01     Sekundäradresse laden
C062 JSR $FFBA    File-Parameter setzen
C065 LDA $02      Byte-Anzahl des Filenamens laden
C067 LDX #$D0     Position des Filenamens
C069 LDY #$C0     laden
C06B JSR $FFBD    Filenamen-Parameter setzen
C06E LDX #$01     Startadresse
C070 LDY #$08     laden
C072 STX $FB      und
C074 STY $FC      speichern
C076 LDA #$FB     Zeiger auf Startadresse laden
C078 LDX $2D      Endadresse
C07A LDY $2E      laden
C07C JSR $FFD8    SAVE-Befehl
C07F JMP $A533    BASIC-Zeilen neu binden und Rücksprung


C080 33 A5 FF FF 8F 20 22 A2   Erste BASIC-Zeile
```

Maschinenprogramm hinter REM-Zeile:

```
C087 LDX #$01     LOW-Byte der Zeilennummer
C089 STX $0803    setzen
C08C DEX          Byte auf Null setzen
C08D STX $0804    HIGH-Byte der Zeilennummer speichern
C090 LDA #$1C     Vektor für Eingabe einer Zeile
C092 STA $0302    auf
```

```
C095 LDA #$08    $081C
C097 STA $0303   stellen
C09A RTS         Rücksprung
C09B LDX #$FF    Zeilennummer der REM-Zeile
C09D STX $0803   auf
C0A0 STX $0804   65535 setzen
C0A3 LDA #$83    Vektor
C0A5 STA $0302   auf
C0A8 LDA #$A4    Originalwert
C0AA STA $0303   stellen
C0AD JMP $A483   Befehl ausführen und Rücksprung


C0B0 00 3C 08 02 00 9E 32 30   Zweite
C0B8 35 36 00 00 00 00 00 00   BASIC-Zeile


C0C0 50 52 4F 47 52 41 4D 4D PROGRAMM
C0C8 4E 41 4D 45 3A 20 00 00 NAME:
```

Nachfolgend nun der entspechende BASIC-Loader:

```
100 FORI=1TO208STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120                     A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A:NEXT:READA:
IFC=ATHENC=0:NEXT:SYS 49152
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA A2,00,BD,80,C0,9D,01,08,E8,E0,3B,D0,F5,20,44,  113
301 DATA E5,A2,00,BD,C0,C0,20,D2,FF,E8,E0,0E,D0,F5,A2,  242
302 DATA 00,20,CF,FF,9D,D0,C0,E8,C9,0D,D0,F5,CA,86,02,  240
303 DATA A2,08,A0,00,20,BA,FF,A5,02,A2,D0,A0,C0,20,BD,  121
304 DATA FF,A9,00,A2,3C,A0,08,20,D5,FF,A5,AE,85,2D,A5,  204
305 DATA AF,85,2E,A6,02,A9,2F,9D,D0,C0,E8,A9,41,9D,D0,  78
306 DATA C0,E8,86,02,A2,08,A0,01,20,BA,FF,A5,02,A2,D0,  109
307 DATA A0,C0,20,BD,FF,A2,01,A0,08,86,FB,84,FC,A9,FB,  44
308 DATA A6,2D,A4,2E,20,D8,FF,4C,33,A5,FF,FF,8F,20,22,  143
309 DATA A2,01,8E,03,08,CA,8E,04,08,A9,1C,8D,02,03,A9,  160
310 DATA 08,8D,03,03,60,A2,FF,8E,03,08,8E,04,08,A9,83,  251
```

```
311 DATA 8D,02,03,A9,A4,8D,03,03,4C,83,A4,00,3C,08,02, 43
312 DATA 00,9E,32,30,35,36,00,00,00,00,00,00,50,52,4F, 92
313 DATA 47,52,41,4D,4D,4E,41,4D,45,3A,20,00,00,00,00, 239
```

2.3 RESET- und RUN-STOP/RESTORE-Schutz

Gegen einen RESET gibt es nur eine einzige Schutzmöglichkeit, und zwar mit CBM80.

CBM80 ist nichts weiter als ein Erkennungsmerkmal, das vom Betriebssystem benutzt wird, um festzustellen, ob sich ein Modul im Modul-Schacht befindet.

Die Byte-Folge CBM80 darf nur im Bereich $8004 (32772) stehen, da es an anderer Stelle nicht anerkannt wird. Von $8000 bis $8003 stehen noch vier Bytes, die in LOW- und HIGH-Byte Darstellung auf die Adresse zeigen, wo der Einsprung, nach Drücken von RESTORE oder nach der Ausführung von RESET stattfinden soll.

Die Bytes in $8000 und $8001 sind für die RESTORE-Taste bestimmt. Nach Drücken von RESTORE springt das Betriebssystem über diese Vektoren in den gewünschten Bereich. Die Vektoren in $8002 und $8003 sind für den RESET bestimmt. Falls ein RESET ausgeführt wird, springt das Betriebssystem eben über die Vektoren in diesen bestimmten Bereich. Da man aber ein Modul simulieren kann, indem man diese Bytes in den Bereich von $8000 bis $8008 hineinschreibt, kann dadurch ein sehr guter RESET- und RUN-STOP/RESTORE-Schutz erzielt werden.

Wir haben hier für Sie ein Programm vorbereitet, das wieder in ein BASIC-Programm springt und es erneut startet, falls RESET oder RESTORE ausgeführt wurden. Es eignet sich gut als Schutz-Methode, da man nicht ohne weiteres aus dem BASIC-Programm herauskommt.

Hier nun das Programm im Assemblerlisting und der BASIC-Loader:

```
8000: 09 80 09 80 C3 C2 CD 38   CBM80 UND
8008: 30 . . . . . .            SPRUNGVEKTOREN NACH $8009

8009 CLI          Interrupt freigeben
800A JSR $FF5B    Video-RESET ausführen
800D JSR $FDA3    Interrupt vorbereiten
8010 LDA #$EF     STOP-Taste
8012 STA $0328    sperren
8015 JSR $A659    CHRGOT-Zeiger auf Programmstart und CLR
8018 JMP $A7AE    in die Interpreter-Schleife springen
```

Dazu der BASIC-Loader:

```
5 POKE 808,225:POKE 56,128:CLR
10 FOR I=0 TO 26: READ A: POKE 32768+I,A: S=S+A: NEXT I
20 IF S<>3182 THEN PRINT "FEHLER IN DATA-ZEILEN": END
100 DATA 9,128,9,128,195,194,205,56,48,88,32,91,255,32,163,253
110 DATA 169,239,141,40,3,32,89,166,76,174,167
```

Der erste POKE-Befehl in Zeile 5 sperrt die RUN/STOP-Taste, weil vor dem Drücken von RUN/STOP-RESTORE oder RESET noch kein Schutz vorhanden ist und man das BASIC-Programm ohne weiteres unterbrechen kann.

Der nachfolgende Befehl setzt das Variablenende in $8000 ab, weil bei längeren Programmen das CBM80 und die Maschinenroutine von den Variablen überschrieben werden können. Dadurch hat man aber erheblich weniger Speicherplatz, weil normalerweise der BASIC-Speicherplatz bis $A000 reicht.

Das Programm funktioniert folgendermaßen: von $8000 bis $8008 sind das CBM80 und die Zeiger, die auf das kleine Programm zeigen, abgelegt. Falls ein RESET oder ein RESTORE ausgeführt wird, springt das Betriebssystem über die Vektoren in das kleine Assemblerprogramm. Dort passiert dann folgendes:

In der ersten Zeile wird der Interrupt, der bei einem RESET maskiert wird, wieder freigegeben. Anschließend wird ein Video-RESET ausgeführt. Das heißt, der VIDEO-Controller wird wieder in den Normalzustand versetzt. Diese Funktion ist nicht unbedingt notwendig und kann auch weggelassen werden. Wir haben sie hier nur eingebaut, weil unter anderem auch der Bildschirm gelöscht wird.

In der nächsten Zeile wird die Betriebssystem-Routine 'INTERRUPT VORBEREITEN' aufgerufen. Das ist nötig, weil sonst nach dem Drücken des RESET-Knopfes keine Eingaben von der Tastatur mehr möglich wären und daher auch keine INPUT-Befehle im BASIC-Programm funktionieren würden.

In den nächsten beiden Zeilen wird die STOP-Taste gesperrt, da man sonst das BASIC-Programm einfach mit der STOP-Taste wieder unterbrechen könnte. Das geschieht, indem das LOW-Byte des STOP-Vektors um einen Befehl im Betriebssystem heruntergesetzt wird, damit diese Routine die STOP-Taste nicht mehr abfragen kann.

Anschließend wird der CHRGET-Zeiger auf das BASIC-Programm gestellt und ein CLR ausgeführt. Dieses muß deshalb geschehen, weil in der letzten Zeile in die Interpreterschleife gesprungen wird. Das heißt, es wird ein RUN ausgeführt.

Wenn Sie nun ein BASIC-Programm mit diesem Programm und einem Autostart ausrüsten, wird man nicht in der Lage sein, aus dem Programm wieder herauszukommen, es sei denn, man schaltet den Computer aus oder arbeitet mit einem anderen Betriebssystem, in dem man durch gleichzeitiges Drücken einer Tastenkombination und der RESET-Taste diese Markierung (CBM80) übergehen kann.

2.4 Warum eigentlich Compilieren?

Ein Compiler ist dazu gedacht, BASIC-Programme schneller zu machen. Da er aber beim Compilieren den BASIC-Code in einen

fast unleserlichen Code umwandelt, eignet sich ein Compiler sehr gut zum Schützen von Programmen.

Es ist ratsam, die LADE-Programme zu compilieren, die einen Hauptteil nachladen, der nachher zum Beispiel noch ein Password abfragt, damit Unbefugte nicht einfach an das Password herankommen oder die Password-Abfrage herausbauen können.

Natürlich sollte man Passwords nicht einfach als ASCII-Codes in seinem Programm ablegen, weil man sonst trotz des Compilers das Password hinterher mit einem Maschinensprachemonitor sehen könnte. Dasselbe gilt auch für Disketten-Befehle, die an die Floppy gesandt werden.

Passwords oder Floppy-Befehle sollte man mit Hilfe des CHR$-Befehls in sein BASIC-Programm einbauen. Dieses hat den Vorteil, daß die Befehle auch in diesem Compiler-Code übersetzt werden und nur mit dem Entschlüsselungsalgorithmus des Compilers erkannt werden können.

2.4.1 Warum der Compilercode keinen absoluten Schutz bildet!

Das, was ein Compiler aus einem BASIC-Programm macht, ist zwar extrem schwer zu entschlüsseln, aber findige Knacker schaffn es, auch solchen Code zu lesen. Das liegt daran, daß der Compilercode ursprünglich gar nicht speziell dafür gedacht war, Programme zu schützen, sondern nur dafür, BASIC-Programme zu beschleunigen.

Es gibt prinzipiell zwei Arten von BASIC-Compilern. Die erste Art erzeugt einen sogenannten P-Code, die zweite direkt einem Maschinencode. P-Code ist ein Code, der erst während des Programmablaufs noch von einem Interpreter übersetzt werden muß, ähnlich wie ein BASIC-Programm vom BASIC-Interpreter ausgeführt wird.

Im Gegensatz zu BASIC-Befehlen sind die Operationen, die ein P-Code-Befehl bewirkt, weitaus simpler, also der Maschinen-

sprache näher, als der entsprechende BASIC-Befehl. Daher sind solche P-Code-Programme auch schneller als BASIC-Programme. Ein Beispiel:

```
POKE 53280,0
```

wird vom BASIC-Interpreter folgendermaßen ausgeführt: Zuerst wird der Befehl POKE, der als Token abgelegt ist, erkannt. Der Interpreter weiß nun, daß zwei Zahlen folgen müssen, die im Programmspeicher als ASCII-Codes stehen und durch ein Komma getrennt werden. Die beiden Zahlen werden geholt, vom ASCII-Format ins Fließkommaformat umgewandelt und schließlich vom Fließkommaformat ins Integer-Format.

Dabei muß außerdem geprüft werden, ob die Syntax des Befehls richtig ist und ob die Parameter zulässige Werte besitzen. Dann erst kann die eigentliche Operation ausgeführt werden. Wenn ein solcher Befehl in den P-Code übersetzt wurde, weiß der P-Code-Interpreter, daß die Syntax des entsprechenden P-Code-Befehls richtig ist.

Außerdem werden die Zahlen gleich im richtigen Format hinter dem Befehl abgelegt. Daher wird der Befehl wesentlich schneller ausgeführt als der ursprüngliche BASIC-Befehl. Er ist allerdings nicht so schnell wie die direkte Sequenz in Maschinensprache:

```
LDA #$00
STA $D020
```

Weiterhin ersetzt ein Compiler Variable und Zeilennummern durch ihre Speicheradressen. Maschinencode-Compiler unterscheiden sich von P-Code-Compilern meist nur dadurch, daß die Unterprogrammaufrufe, die ein spezieller P-Code bewirken würde, bei ihnen durch direkte Unterprogrammaufrufe mit dem Befehl "JSR" ersetzt werden. Das bringt natürlich nur noch eine leichte Geschwindigkeitssteigerung.

Es gibt auch Compiler, die einen verringerten BASIC-Befehlssatz in nahezu "echte" Maschinensprache übersetzen können. Solche Compiler eignen sich aber meistens nicht zum Schützen, da ihr Code einfach zu verstehen ist.

Einige "Freaks" haben sich allerdings die Arbeit gemacht, den P-Code einiger BASIC-Compiler aufzuschlüsseln. Es existieren sogar für bestimmte Compiler sogenannte RE-Compiler, die den P-Code wieder in BASIC übersetzen.

Sollten Sie Ihr Programm also wirksam mit einem Compiler schützen wollen, so erkundigen Sie sich vorher, ob ein solcher RE-Compiler für Ihren speziellen Compiler auf dem Markt erhältlich ist.

3. Programmschutz für Profis

3.1 Autostart

Sicher wird Ihnen schon einmal aufgefallen sein, daß fast alle professionellen Programme, die es für den C64 zu kaufen gibt, mit einem Autostart versehen sind. Warum sich die Softwarefirmen soviel Mühe geben, einen neuen und komplizierten Autostart zu konzipieren und zu installieren, ist einfach zu beantworten: Das selbständige Einladen des Programms stellt einen Kopierschutz dar, der das Programm vor unerlaubtem Kopieren schützen soll.

3.1.1 Warum eigentlich Autostart?

Bevor man sich über die Technik des Autostarts Gedanken macht, sollte die Frage nach dem Sinn des selbständigen Programmstarts geklärt werden. Soll das Programm mit einem Autostart versehen werden, damit sich der Benutzer nach dem Laden das Eintippen der drei Buchstaben "RUN" ersparen kann, oder soll damit noch ein anderer Zweck erfüllt werden? Ist der erste Grund ausschlaggebend, wird es besser sein, sich auf die einfacheren Methoden zu beschränken, da diese mit wenig Aufwand zu realisieren sind. Außerdem ist der Programmcode relativ kurz und die Ausführung im Vergleich zu den etwas komplizierteren Methoden wesentlich schneller.

Als "Alternative" läßt sich der Autostart aber auch zum Schutz eigener Programme einsetzen. Diese zweite, weitaus interessantere Variante erfordert jedoch einige Vorkenntnisse im Bereich der Assemblerprogrammierung und der Speicheraufteilung des C64. Ist das Prinzip aber einmal erkannt, stehen eine Menge Möglichkeiten zur Verfügung, unbefugten Personen den Einblick in seine Programme zu versperren. Schließlich ist der beste Kopierschutz sinnlos, wenn er sich durch das Löschen weniger BASIC-Zeilen entfernen läßt.

Der Autostart stellt also nicht nur einen "Mini-Kopierschutz" dar, sondern dient vor allem dem Sichern der Kopierschutzabfrage in den einzelnen Programmen. Zusammen mit einer sinnvollen Codierroutine läßt sich dadurch der Zugriff für fast jeden Unbefugten versperren. Hundertprozentig sicher ist aber kein Autostart, da man jeden rückgängig machen kann, wenn das Programm in einen anderen Speicherbereich geladen und so der Autostart unterdrückt wird. Es liegt also vor allem beim Programmierer, hier kreativ zu sein und sich neue Tricks einfallen zu lassen, die zum Beispiel das Abarbeiten eines Programms nur in einem bestimmten Speicherbereich erlauben. Auf gar keinen Fall darf ein Assemblerprogramm mit Autostart relokatibel (an jeder Stelle des Speichers lauffähig) sein. Doch bevor wir tiefer in das Prinzip des Autostarts einsteigen, noch einige Tips zu einfacheren Methoden.

3.1.2 Der einfachste Autostart

Der wohl einfachste Autostart, der sozusagen "serienmäßig" im C64 eingebaut ist, läßt sich durch die Tastenkombination "SHIFT-RUN/STOP" erreichen. Es erscheint "LOAD" und zwei Zeilen tiefer "PRESS PLAY ON TAPE" auf dem Bildschirm. Für die Besitzer einer Datasette bedeutet diese Tastenkombination, daß das nächste Programm von der Kassette geladen und sofort danach gestartet wird. Ein Autostart erfolgt natürlich nur, wenn das Programm auch sonst mit dem BASIC-Befehl RUN gestartet werden kann.

Der Benutzer einer Diskettenstation kann mit diesen Tasten aber nichts anfangen, da der Parameter ",8", der für das Laden von der Floppy unbedingt nötig ist, nicht auftaucht.

Aber auch hier kann mit der oben erwähnten Tastenkombination etwas erreicht werden. Tippen Sie zuerst die Befehlszeile zum Laden des Programms ein, die etwa so aussieht:

```
LOAD "PROGRAMMNAME",8
```

Doch anstatt die RETURN-Taste zu drücken, schreiben Sie ":" und drücken jetzt "SHIFT-RUN/STOP". Nach dem Laden erscheint "RUN" auf dem Bildschirm, und das Programm wird gestartet.

3.1.3 Autostart im Tastaturpuffer

Die oben besprochene Methode zum automatischen Programmstart bezieht sich nur auf das Starten selbst, indem das Eingeben des Befehls RUN "simuliert" wird; der Programmcode wird dabei allerdings nicht geändert. Diese Methode ist also nicht dauerhaft und hat nichts mit Programmschutz zu tun.

Interessanter ist die Möglichkeit, den Autostart mit dem eigentlichen Programm abzuspeichern, so daß dieses nach jedem Laden gestartet wird. Ein einfacher Trick besteht darin, einen Befehl auf den Bildschirm zu schreiben und dann das Drücken der RETURN-Taste zu simulieren. Um das Prinzip zu verstehen, geben Sie folgende Zeile ein:

```
PRINT "(CLR/HOME) PRINT 3 + 4"
```

Durch die Eingabe dieser Zeile wird der Bildschirm gelöscht, und es erscheint der Text "PRINT 3 + 4" am oberen Bildschirmrand. Bis jetzt hat sich noch nichts ereignet, was an die Automatisierung eines Befehls oder gar eines Programmstarts erinnern würde. Doch gehen Sie jetzt mit dem Cursor einige Zeilen nach unten und geben die folgende Zeile ein:

```
POKE 631,19:POKE 632,13:POKE 198,2
```

Wenn Sie beim Eintippen der beiden Zeilen keine Fehler gemacht haben, wird die Zeile, die am oberen Bildschirmrand abgebildet ist, ausgeführt, und der Cursor erscheint zusammen mit der READY-Meldung zwei Zeilen tiefer. Es besteht also kein Unterschied zu einem Befehl, den Sie normalerweise im Direktmodus eingeben.

Schauen wir uns den Vorgang einmal genauer an, stellen wir fest, daß der Cursor in die erste Zeile gebracht und der dort stehende Befehl ausgeführt wurde. Wir haben den Computer also dazu veranlaßt zu erkennen, daß die HOME- und daraufhin die RETURN-Taste gedrückt wurde. Die ganze Aktion wurde natürlich durch die drei zuletzt eingegebenen POKEs veranlaßt. Untersuchen wir die angesprochenen Speicherzellen und die Werte, die in diese geschrieben werden, läßt sich das "Phänomen" relativ einfach erklären.

Durch POKE 631,19 und POKE 632,13 werden die Werte für gedrückte Tasten in den sogenannten Tastaturpuffer ab $0277 (631) geschrieben. In diesem Tastaturpuffer werden normalerweise die Tastencodes zwischengespeichert, die zwar vom Benutzer schon eingegeben wurden, aber noch nicht verarbeitet werden konnten. Das Betriebssystem liest diese Codes in regelmäßigen Abständen aus und gibt sie an den BASIC-Interpreter weiter. Mit unseren POKEs haben wir die Codes für die CLR- und die RETURN-Taste in diesen Puffer geschrieben. Nun muß dem Betriebssystem noch die Anzahl der gedrückten Tasten mitgeteilt werden, was durch POKE 198,2 geschieht. In der Adresse $C6 (198) befindet sich die Anzahl der Zeichencodes, die im Tastaturpuffer gespeichert sind. Da wir zwei Tastencodes in den Puffer geschrieben haben, müssen wir hier den Wert 2 ablegen. Da der Inhalt des Puffers regelmäßig verarbeitet wird, haben wir so indirekt die Ausführung eines Befehls veranlaßt.

Um nun unsere eigentliche Aufgabenstellung, den selbständigen Programmstart, zu lösen, muß eine Methode gefunden werden, den Befehl RUN unmittelbar nach dem Laden ausführen zu lassen. Dazu benutzen wir das oben verwandte Prinzip, mit dem Unterschied, daß die oberste Zeile "PRINT 3 + 4" durch "RUN" ersetzt wird. Die POKEs, die die Tastencodes in den Tastaturpuffer schreiben, können wir unverändert übernehmen, da diese unabhängig vom Inhalt der obersten Bildschirmzeile sind. Daraus folgt, daß auch POKE 198,2 übernommen wird.

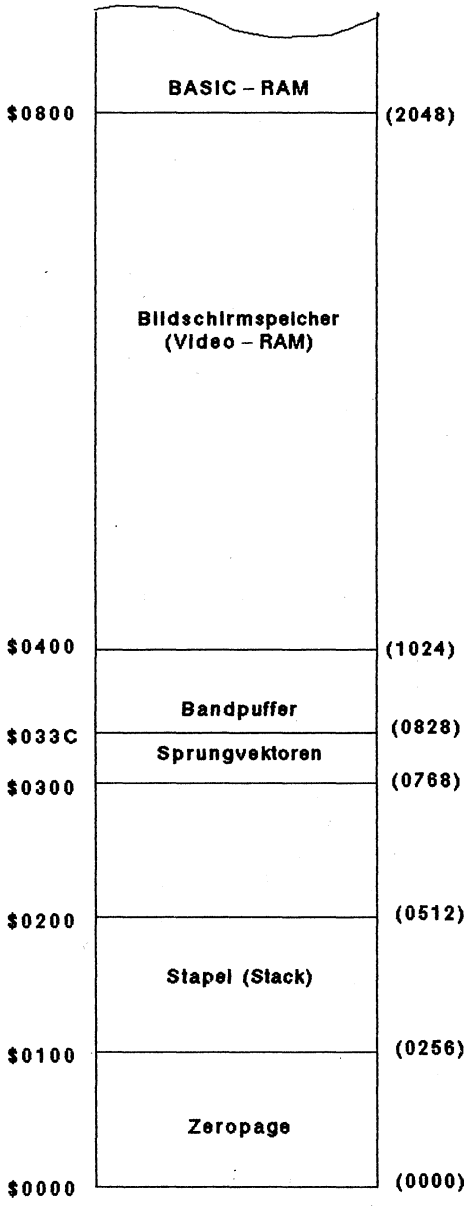


Abb. 3.1.1: Unterer Speicherbereich des C64

Das Problem besteht nun darin, den Inhalt des Tastaturpuffers und dessen Zähler zusammen mit dem Hauptprogramm abzuspeichern, so daß die Inhalte nach jedem Laden vorhanden sind, um den Autostart auszuführen. Schauen wir uns dazu die Abbildung 3.1.1 an, in der der untere Speicherbereich des C64 abgebildet ist. Wie dort zu sehen ist, beginnt das BASIC-RAM ab $0800 (2048). Da in der Speicherzelle $0800 (2048) immer $00 steht, werden alle BASIC-Programme ab $0801 (2049) geladen und auch abgespeichert. Bestimmt wird die Anfangs- und Endadresse von BASIC-Programmen durch Vektoren in der Zeropage. Die Anfangsadresse ist in $2B/$2C (43/44) und die Endadresse in $2D/$2E (45/46) im LOW- und HIGH-Byte-Format abgelegt. LOW- und HIGH-Byte-Format bedeutet hierbei, daß zuerst das LOW-Byte und dann erst das HIGH-Byte abgespeichert wird. Die BASIC-Startadresse ist also folgendermaßen abgelegt:

```
$002B 01 08 .. .. ..
```

Setzt man die BASIC-Startadresse auf $C6 (198), wird durch

```
SAVE "PROGRAMMNAME",8
```

der gesamte Speicherbereich von $C6 (198) bis zum Programmende abgespeichert, inklusive des Tastaturpuffers und dessen Zähler. Wenn wir dazu die oben erwähnten POKEs eingeben und sich das Hauptprogramm zu diesem Zeitpunkt im Speicher befindet, wird nach jedem Laden ein Autostart durchgeführt. Zu beachten ist noch, daß die Zeiger auf das Programmende dabei neu gesetzt werden müssen. Wir lesen diese also zuvor durch PEEK (45) und PEEK (46) aus und schreiben sie mit in die oberste Bildschirmzeile.

Der gesamte Ablauf sieht nun also wie folgt aus: Laden Sie das mit einem Autostart zu versehende BASIC-Programm und tippen Sie als erstes

```
PRINT"(CLR/HOME)POKE 45,"PEEK(45)":POKE 46,"PEEK(46)":RUN"
```

ein. Nach dem Drücken der RETURN-Taste sollte der Bildschirm gelöscht werden und

```
POKE45,(Wert1):POKE46,(Wert2):RUN
```

in der ersten Bildschirmzeile erscheinen. Wert1 und Wert2 stehen hier symbolisch für die BASIC-Endadresse, die ja, wie oben schon erwähnt, in den Speicherzellen 45 und 46 im LOW- und HIGH-Byte-Format abgelegt ist.

Gehen Sie nun mit den Cursor zwei Zeilen tiefer und geben wieder

```
POKE 631,19:POKE 632,13:POKE 198,2:
POKE 43,198:POKE44,0:SAVE "NAME",8
```

ein. Nun wird das im Speicher befindliche Programm auf Diskette abgespeichert, nachdem die Startadresse durch POKE 43,198 und POKE 44,0 auf $C6 (198) heruntergesetzt wurde. Geben Sie jetzt POKE 43,1:POKE44,8 ein, wodurch die Startadresse wieder auf ihren Normalwert zurückgesetzt wird und Sie wie gewohnt weiterarbeiten können.

Das Programm befindet sich nun in einer etwas längeren Version auf Diskette und kann mit LOAD "NAME",8,1 geladen werden. Geben Sie aber vor dem Laden bitte NEW oder CLR ein, da sich sonst Probleme mit dem (mitabgespeicherten) Stapel ergeben können. Da wir, wie aus Abbildung 3.1.1 ersichtlich, auch den Bildschirmspeicher mit gespeichert haben, wird der Bildschirm beim Laden überschrieben, und es erscheinen die POKEs und der RUN-Befehl. Zusätzlich zu diesen POKEs können Sie auch noch weitere in die erste Zeile schreiben, bevor Sie das Programm abspeichern. Zum Beispiel wird durch POKE 808,225 der STOP-Vektor so verändert, daß die STOP-Taste nicht mehr abgefragt wird und so ein Anhalten Ihres Programms durch RUN/STOP nicht mehr möglich ist.

3.1.4 Autostart über Sprungvektoren

Die oben besprochene Autostartmethode zeichnet sich duch ihre verblüffende Einfachheit aus. Dem BASIC-Programmierer wird damit die Möglichkeit geboten, mit geringem Aufwand und minimalen Vorkenntnissen über die Speicheraufteilung des C64 seine eigenen Programme mit einem Autostart zu versehen. Diese Methode ist jedoch mit einigen Nachteilen verbunden. Beispielsweise ist die Eingabe von NEW vor dem Laden nicht wünschenswert. Weiterhin kann man beim Einladen auf dem Bildschirm sehen, mit welchen Befehlen der Autostart erzeugt wurde. Dazu kommt noch, daß im Vergleich zu anderen Verfahren das bearbeitete Programm relativ viele Blöcke mehr benötigt als ohne den Autostart.

Die nun folgenden Autostartmethoden beziehen sich hauptsächlich auf Assemblerprogramme, was nicht bedeutet, daß mit ihnen nicht auch BASIC-Programme gestartet werden können.

Einleitend muß erwähnt werden, daß der C64 im Gegensatz zu vielen anderen Computern über ein ROM (Read Only Memory) verfügt, in dem das Betriebssystem des Rechners fest abgelegt ist. Das hat den Vorteil, daß das Betriebssystem nach dem Einschalten nicht erst "gebootet", also eingeladen, werden muß. Andererseits bringt gerade diese Eigenschaft den großen Nachteil für den Anwender mit sich, daß er das ROM nicht abändern und seinen Bedürfnissen anpassen kann. Um dem ein wenig abzuhelfen, bietet der C64 eine Reihe von sogenannten Sprungvektoren an, die zwar vom Betriebssystem benutzt werden, aber nicht im ROM, sondern am Anfang des Arbeitsspeichers abgelegt sind. Diese Vektoren liegen ab der Adresse $0300 (768) und verweisen auf die jeweiligen Routinen innerhalb des Betriebssystems. An dieser Stelle kann der Programmierer eingreifen, indem er die Vektoren auf seine eigenen Routinen stellt. Es lassen sich jedoch auch andere Effekte erreichen, die äußerst nützlich sein können.

Hier soll eine schematische Darstellung verdeutlichen, wie die Sprungvektoren vom Betriebssystem benutzt werden.

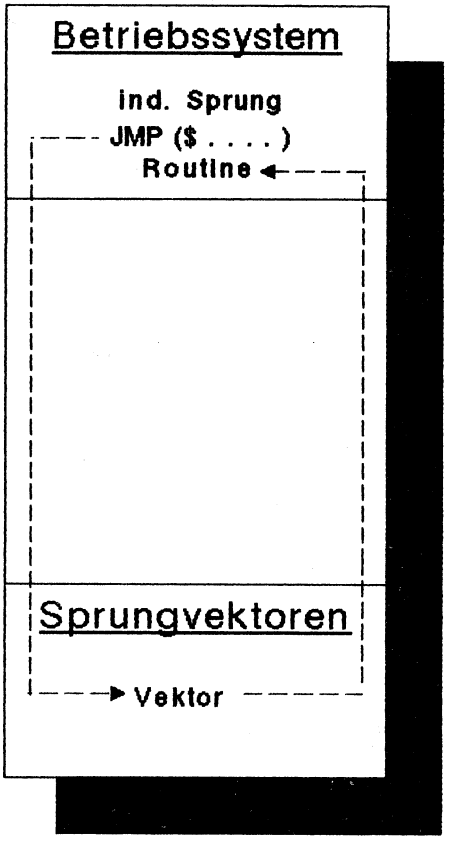


Abb. 3.1.2: Funktion der Sprungvektoren

Nachdem eine Routine direkt oder aus einer anderen Routine heraus angesprungen wurde, wird durch einen indirekten Sprung über die Vektoren zur eigentlichen Routine verzweigt. Diese Routine befindet sich im allgemeinen direkt hinter dem indirekten Sprungbefehl. Die ersten Vektoren sind die des BASIC-Interpreters, die die nachfolgenden aufgeführten Funktionen

erfüllen. Zum besseren Verständnis ist es ratsam, die Routinen in einem ROM-Listing, wie es zum Beispiel in dem Buch "64 INTERN" abgedruckt ist, nachzuschlagen.

Adresse	Vektor	Beschreibung
$0300/0301 (768/769)	$E38B	Vektor für BASIC-Warmstart; wird nach END sowie beim Auftreten eines Fehlers angesprungen (Fehlernummer im Akku).
$0302/0303 (770/771)	$A483	Vektor für Eingabe einer Zeile; Rechner bleibt in der Eingabewarteschleife, bis RETURN erfolgt.
$0304/0305 (772/773)	$A57C	Vektor für Umwandlung in den Interpretercode.
$0306/0307 (774/775)	$A71A	LIST-Vektor; wird bei Umwandlung in den Klartext angesprungen.
$0308/0309 (776/777)	$A7E4	Vektor für BASIC-Befehlsadresse holen; zeigt an die Stelle des Interpreters, die den BASIC-Befehl ausführt.
$030A/030B (778/779)	$AE86	Vektor wird angesprungen, wenn ein Element eines Ausdrucks berechnet werden soll.
$0311/0312 (785/786)	$B248	USR-Vektor; steht normalerweise auf 'ILLEGAL QUANTITY'.

Die Vektoren des Betriebssystems stehen ab $0314/0315

Adresse	Vektor	Beschreibung
$0314/0315 (788/789)	$EA31	IRQ-Vektor; wird jede 1/60 Sekunde angesprungen.
$0316/0317 (790/791)	$FE66	BRK-Vektor
$0318/0319 (792/793)	$FE47	NMI-Vektor; wird beim Drücken der RESTORE-Taste benutzt.
$031A/031B (794/795)	$F34A	OPEN-Vektor
$031C/031D	$F291	CLOSE-Vektor

```
(796/797)
$031E/031F  $F20E    CHKIN-Vektor
(798/799)
$0320/0321  $F250    CKOUT-Vektor
(800/801)
$0322/0323  $F333    CLRCH-Vektor
(802/803)
$0324/0325  $F157    INPUT-Vektor; zeigt normalerweise auf
(804/805)            Routine für Eingabe von der Tastatur.
$0326/0327  $F1CA    OUTPUT-Vektor; zeigt normalerweise auf
(806/807)            Routine für Ausgabe auf den Bildschirm.
$0328/0329  $F6ED    STOP-Vektor
(808/809)
$032A/032B  $F13E    GET-Vektor
(810/811)
$032C/032D  $F32F    CLALL-Vektor
(812/813)
$032E/032F  $FE66    Warmstart-Vektor
(814/815)
$0330/0331  $F4A5    LOAD-Vektor
(816/817)
$0332/0333  $F5ED    SAVE-Vektor
(818/819)
```

Mit diesen Vektoren lassen sich nun einige 'Kunststücke' vollbringen. Man kann zum Beispiel die LIST-Vektoren so umstellen, daß diese auf ein RTS zeigen. Das bewirkt, daß nach einem LIST-Befehl sofort wieder ins BASIC zurückgesprungen wird, ohne daß überhaupt etwas gelistet wird. Der Befehl wird also einfach ignoriert. Wir haben noch einige andere Ideen für Sie aufgeschrieben:

POKE 774,	POKE775,	Adresse	Routine
226	252	$FCE2 (64738)	RESET
68	166	$A644 (42564)	NEU
7	168	$A807 (43015)	SYNTAX ERROR
160	240	$F0A0 (61600)	Absturz des Systems

Natürlich lassen sich auch die anderen Vektoren auf diese Routinen 'umbiegen'. Wirkungsvoll wäre es zum Beispiel noch beim SAVE-Vektor. Das Umstellen dieses Vektors läßt zu, daß man mit einem Programm wie üblich arbeiten kann; man kann jedoch danach nicht wieder abspeichern. Dieser Trick stellt also einen kleinen Kopierschutz dar.

Für den BASIC-Anwender kann es auch von Nutzen sein, den RESTORE-Vektor abzuändern, wodurch eine Unterbrechung des Programms mittels RUN/STOP-RESTORE unmöglich wird. Auch hier kann wieder zu einer eigenen Routine verzweigt, bzw. die Funktion ganz ausgeschaltet werden. Dies kann zum Beispiel durch POKE 792,193:POKE 793,254 erzielt werden.

Durch das Drücken der RESTORE-Taste wird eine bestimmte Leitung auf Masse gelegt, wodurch ein NMI (nicht maskierbarer Interrupt) ausgelöst wird. Bei einem NMI wird automatisch zur Adresse $FFFA verzweigt, was mit einer bestimmten Eigenart des Prozessors zusammenhängt. Von dort aus wird die NMI-Routine angesprungen, die sofort einen indirekten Sprung über den oben genannten Vektor ausführt. Durch die oben genannten POKEs wird die Zieladresse des NMI-Vektors in $0318/0319 (792/793) von $FE47 auf $FEC1 umgeändert. In der neuen Zieladresse steht aber ein RTI-Befehl (Return from Interrupt), so daß unmittelbar nach dem Aufruf des NMI ein Rücksprung erfolgt.

Nun zu einer interessanteren Anwendung dieser Vektoren, dem Autostart. Wie Sie sicherlich wissen, kann man mit LOAD"PROGRAMMNAME",8,1 ein Programm in einen bestimmten Bereich des Speichers laden, wenn es zuvor aus diesem Bereich heraus abgesichert wurde. Dieses kann man am einfachsten mit einem Maschinensprachemonitor, wie zum Beispiel dem PROFI-MON, machen. Werden nun beim Laden eines Programms die Sprungvektoren überschrieben, werden dadurch alle Zeiger umgestellt, und der Rechner stürzt ab. Wurde jedoch vor dem Abspeichern nur ein Zeiger umgeändert und die anderen in ihrem Zustand gelassen, lädt der Computer ordnungsge-

mäß, und nur eine einzige Routine wird anders ausgeführt. Man kann nun einen Vektor umstellen, der nur manchmal benutzt wird, zum Beispiel den LIST-Vektor, oder einen, über den ständig verzweigt wird. Ein Vektor, über den immer verzweigt wird und der deshalb ideal erscheint, ist der STOP-Vektor in $0328/0329 (790/791). Auch während des Ladens wird über diese Adresse gesprungen, so daß schon vor Beendigung des Ladevorgangs ein Autostart erfolgen kann. Doch die speziellen Eigenarten dieser Routine wollen wir später noch genauer betrachten.

Für unseren normalen Autostart bietet sich auch die Eingabewarteschleife ($0302/0303) an, die fast ständig auf eine Eingabe von der Tastatur wartet. Diesen Vektor kann man nun auf den eigenen Programmanfang stellen, so daß nach dem Laden nicht mehr der Cursor erscheint, sondern direkt ein Programmstart erfolgt. Wie aus Abbbildung 3.1.1 ersichtlich, ist es sinnvoll, das Programm in den Kassettenpuffer ab $033C (828) zu legen, da sonst der Bildschirm mit abgespeichert werden muß.

Auf diese Weise kann aber nur ein Maschinenprogramm gestartet werden, da zu einer bestimmten Adresse gesprungen wird. Soll ein BASIC-Programm gestartet werden, so muß man eine kleine Routine zwischenschalten, die folgendermaßen aussieht:

```
033C JSR $A659 ;CHRGET-Zeiger auf Programmstart und CLR
033F JMP $A7AE ;in die Interpreterschleife springen
```

Eine andere Möglichkeit ist es, über den OUTPUT-Vektor $0326/0327 (806/807) zu verzweigen, da dieser bei jeder Ausgabe auf den Bildschirm angesprungen wird und dementsprechend schwer zu unterdrücken ist. Diese Eigenschaft des OUTPUT-Vektors stellt uns jedoch vor ein Problem. Wie soll ein Programm mit dem abgeänderten Vektor geSAVEd werden, ohne daß es schon bei der Ausgabe der Meldung "SAVING" anläuft?

Die einfachste Möglichkeit besteht darin, den Vektor zunächst unverändert zu lassen und das Programm in den gewünschten Bereich, am besten ab $033C (828), zu schreiben. Nach der Fertigstellung des Programms wird dieses mitsamt den Sprungvek-

toren ab $0300 (768) in einen anderen Speicherbereich kopiert. Dieses Kopieren kann durch eine selbstgeschriebene Routine oder mit Hilfe eines Maschinensprachemonitors geschehen. Der Standardbefehl dazu steht in fast jedem Monitor zur Verfügung und hat folgendes Format:

T (Anfangsadresse) (Endadresse) (neue Anfangsadresse)

Das Transferieren sollte möglichst in Schritten von $1000 (4096) vorgenommen werden, da die Adressenumrechnung dadurch erheblich vereinfacht wird. Ein Programm, das im Kassettenpuffer liegt, sollte also möglichst mit den Sprungvektoren ab $0300 in den Bereich ab $1300 verschoben werden.

Danach wird der OUTPUT-Vektor, der sich jetzt in $1326/1327 befindet, auf unseren Programmstart abgeändert. Beginnt das Programm im Orginalbereich ab $033C, so muß diese Adresse in den OUTPUT-Vektor geschrieben werden, was dann so aussieht:

```
$1300 3C 03
```

Beachten Sie dabei, daß die Sprungvektoren im LOW- und HIGH-Byte-Format vorliegen.

Das folgende Programm ermöglicht es Ihnen, ein gewöhnliches Programm, das mit RUN gestartet wird, mit einem Autostart zu versehen.

```
C000 LDA #$00      Farbwert für Schwarz laden und
C002 STA $D020     in Register für Hintergrund und
C005 STA $D021     Vordergrund schreiben
C008 LDA #$05      Farbwert für Grün
C00A STA $0286     in Register für Schriftfarbe
C00D LDX #$00      Zeiger auf $00 stellen
C00F LDA $C090,X   erstes ASCII-Zeichen holen
C012 CMP #$20      mit SPACE vergleichen
C014 BEQ $C01D     wenn SPACE, dann Schleife beenden
C016 JSR $FFD2     Zeichen auf Bildschirm ausgeben
C019 INX           Zeiger um 1 erhöhen
C01A JMP $C00F     nächstes Zeichen holen
C01D JSR $C082     Unterroutine für Eingabe des Pgm-Namens
C020 LDX #$08      Wert für Laufwerk (,8)
C022 LDY #$01      Sekundäradresse = 1 setzen
C024 JSR $FFBA     File-Parameter setzen
C027 LDX #$00      LOW-Byte des Zeigers auf Filenamen
C029 LDY #$CF      HIGH-Byte des Zeigers auf Filenamen
C02B STX $BB       Zeiger auf Filename setzen
C02D STY $BC       (Zeropage 187/188)
C02F LDA #$00      Wert für Programmodus in
C031 STA $9D       Flag schreiben (Meldung unterdrücken)
C033 JSR $FFD5     Programm laden
C036 LDA $90       Wert aus Fehlerstatus laden
C038 CMP #$40      und testen
C03A BNE $C000     Fehler, dann zum Programmstart zurück
C03C LDX #$00      Zeiger auf $00 setzen
C03E LDA $C091,X   ASCII-Zeichen holen
C041 BEQ $C04A     Ende, dann Schleife beenden
C043 JSR $FFD2     Zeichen ausgeben
C046 INX           Zeiger erhöhen
C047 JMP $C03E     nächstes Zeichen holen
C04A JSR $C082     Filenamen eigeben
C04D LDX #$00      Zeiger auf $00 setzen
C04F LDA #$20      Wert für SPACE
C051 STA $0400,X   in Bildschirmspeicher schreiben
C054 INX           Zeiger erhöhen
C055 BNE $C051     kleiner 255, nächste Position löschen
C057 LDX #$40      LOW-Byte des neuen OUTPUT-Vektors
C059 LDY #$03      HIGH-Byte des Vektors
```

```
C05B STX $0326     Werte in OUTPUT-Vektor
C05E STY $0327     schreiben
C061 LDX #$00      Zeiger auf $00 setzen
C063 LDA $C0AA,X   Byte für Startprogramm holen und in
C066 STA $0340,X   Tastaturpuffer schreiben
C069 INX           Zeiger erhöhen
C06A CPX #$10      vergleicht auf Anzahl der Bytes
C06C BNE $C063     nicht alle Bytes, dann weiter
C06E LDX #$00      LOW-Byte und
C070 LDY #$03      HIGH-Byte der Startadresse des Programms
C072 STX $FB       in Zeropage
C074 STY $FC       speichern
C076 LDA #$FB      Zeiger auf Startadresse setzen
C078 LDX $AE       LOW-Byte der Endadresse in X-Register
C07A LDY $AF       HIGH-Byte der Endadresse in Y-Register
C07C JSR $FFD8     Programm speichern
C07F JMP $0340     zu Programmstart (Programm testen)
C082 LDX #$00      Zeiger und Länge des Namens
C084 STX $B7       auf $00 setzen
C086 JSR $FFCF     Zeichen von Tastatur holen
C089 STA $CF00,X   Zeichen speichern
C08C INX           Zeiger erhöhen
C08D INC $B7       Länge des Names erhöhen
C08F CMP #$0D      ist Zeichen = RETURN?
C091 BNE $C086     nein, dann nächstes Zeichen holen
C093 RTS           Rücksprung

C0AA LDX #$CA      LOW-Byte des OUTPUT-Vektors laden
C0AC LDY #$F1      HIGH-Byte des OUTPUT-Vektors laden
C0AE STX $0326     LOW-Byte in OUTPUT-Zeiger schreiben
C0B1 STY $0327     HIGH-Byte in OUTPUT-Vektor schreiben
C0B4 JSR $A659     CHRGET-Zeiger auf Programmstart und CLR
C0B7 JMP $A7AE     in die Interpreterschleife springen


C090 93 0D 50 52 4F 47 52 41     PROGRA
C098 4D 4D 4E 41 4D 45 3A 20     MMNAME:
C0A0 28 4E 45 55 29 00           (NEU)
```

Hier der BASIC-Loader des oben abgedruckten Autostart-Programms:

```
5 N=49152
10 READX:IFX=-1THEN30
20 S=S+X:POKE N,X:N=N+1:GOTO 10
30 IFS<>22926 OR N<>49339 THEN PRINT"FEHLER IN  DATA S":END
40 SYS49152
101 DATA 169,0,141,32,208,141,33,208,169,5,141,134,2,162,0,189,148,192,2
01
102 DATA 32,240,7,32,210,255,232,76,15,192,32,130,192,162,8,160,1,32,186
,255
103 DATA 162,0,160,207,134,187,132,188,169,0,133,157,32,213,255,165,144,
201
104 DATA 64,208,196,162,0,189,149,192,240,7,32,210,255,232,76,62,192,32,
130
105 DATA 192,162,0,169,32,157,0,4,232,208,250,162,64,160,3,142,38,3,140,
39
106 DATA 3,162,0,189,170,192,157,64,3,232,224,16,208,245,162,0,160,3,134
,251
107 DATA 132,252,169,251,166,174,164,175,32,216,255,76,64,3,162,0,134,18
3
108 DATA 32,207,255,157,0,207,232,230,183,201,13,208,243,96,147,13,80,82
,79
109 DATA 71,82,65,77,77,78,65,77,69,58,32,40,78,69,85,41,0,162,202,160,2
41
110 DATA 142,38,3,140,39,3,32,89,166,76,174,167,32,-1
```

3.1.5 Stapel-Autostart

Eine weitere Alternative zu den eben besprochenen Autostartmethoden bietet der sogenannte Stapel-Autostart. Der Stapel, auch Stack genannt, befindet sich im Bereich von $0100 (256) bis $01FF (511) (siehe Abbildung 3.1.1) und enthält unter anderem die Rücksprungadressen des Programms beim Aufrufen von Unterprogrammen. Da auch das Laden von einem Unterprogramm aus geschieht, kann der Stapel dabei mit dem Einsprung unseres Programms überschrieben werden. Dabei muß die Startadresse plus $01 in Low- und High-Byte abgelegt werden.

Beim Rücksprung holt sich der Rechner nun die Adresse aus dem Stapel, die ja mittlerweile abgeändert wurde, und verzweigt dementsprechend.

Genau betrachtet, funktioniert dieses Prinzip nur durch eine besondere Eigenschaft des Stapels: der Stapel ist ein sogenanntes LIFO-Register, das 255 Bytes groß ist. LIFO ist eine Abkürzung für den englischen Ausdruck "LAST IN, FIRST OUT", was soviel bedeutet wie: zuletzt hinein, zuerst heraus. Das heißt also, daß das Byte, das zuletzt auf den Stapel geschoben wurde, als erstes wieder herauskommt. Zu vergleichen ist diese Methode mit einem Stapel von Tellern. Der Teller, der als letztes auf den Tellerstapel gelegt wurde, wird auch als erstes wieder heruntergenommen.

Damit der Computer weiß, welches Byte nun als letztes abgelegt wurde, existiert ein sogenannter Stapelzeiger. Dieser Stapelzeiger ist ein Ein-Byte-Wert, der die "Höhe" des Stapels anzeigt. Steht dieser Zeiger zum Beispiel auf $F7, wird das nächste Byte aus der Adresse gelesen, die sich aus dem Stapelanfang und dem Zeiger ergibt. In diesem Fall wäre das $0100 + $F7 = $01F7.

Während des Programmablaufs kann der Zeiger mit dem TSX-Befehl in das X-Register geschoben und so ausgelesen werden. Umgekehrt kann der momentane Wert des X-Registers mit TXS als Stapelzeiger eingesetzt werden.

Bei einem Unterprogrammaufruf durch den JSR-Befehl (Jump to Subroutine) wird die Adresse des zuletzt bearbeiteten Bytes auf den Stapel geschoben. Folgendes Beispiel soll dieses verdeutlichen:

```
1000 JSR $1080  Aufruf des Unterprogramms
1003 INC $D020  beliebiger nächster Befehl
```

Der Unterprogrammaufruf liegt hier ab der Adresse $1000. Da der JSR-Befehl drei Bytes lang ist, wird die Adresse $1002 auf den Stapel geschoben. Dieses geschieht in der Reihenfolge LOW- und HIGH-Byte. Es wird also zuerst der Wert $02 (LOW-Byte der Adresse) und dann erst der Wert $10 (HIGH-Byte der

Adresse) auf den Stapel geschoben. Bei der Beendigung der Unterprogrammroutine durch den RTS-Befehl (Return from Subroutine) wird die Absprungadresse wieder zurückgeholt. Da der nächste Befehl aber ab $1003 steht, muß zu dem LOW-Byte der abgelegten Adresse noch $01 addiert werden.

Dieses System der Adreßspeicherung im Stapel können wir uns zunutze machen, um nach dem Laden einen Autostart durchführen zu lassen. Wie schon oben erwähnt, werden während des Ladevorgangs im Betriebssystem selbst mehrere Unterprogramme aufgerufen. Zum Beispiel wird die Ausgabe des "SEARCHING FOR" auf dem Bildschirm von einer Unterroutine übernommen. Aber auch der gesamte Ladevorgang wird durch ein RTS beendet. Überschreiben wir nun den gesamten Stapel von $0100 bis $01FF mit bestimmten Werten, wird nicht mehr zu der abgelegten Adresse zurückgesprungen, da diese mittlerweile überschrieben wurde. Es ist auch völlig unerheblich, auf welchen Wert der Stapelzeiger steht, da ja der gesamte Stapel überschrieben wurde.

Jetzt müssen wir uns nur noch überlegen, mit welchen Werten der Stapel überschrieben werden soll. Da der Wert des Stapels unbekannt ist, ist es am einfachsten, den gesamten Stapel mit nur einer Zahl zu füllen. Dafür bietet sich der Wert $02 an. Wenn alle Werte $02 sind, muß bei einem RTS zu der Adresse $0202 + $01, also zu $0203 verzweigt werden.

Schauen Sie sich die Adressen ab $0203 an, so werden Sie feststellen, daß der gesamte Speicherbereich von $0200 (512) bis $0258 (600) als BASIC-Eingabepuffer fungiert. Da wir den BASIC-Eingabepuffer aber momentan nicht brauchen, kann hier ein Programm abgelegt werden. Die nun zur Verfügung stehenden 88 Bytes ($58) reichen zwar nicht für ein längeres Programm, sie bieten aber genug Platz für ein Ladeprogramm, dem eventuell noch eine Decodierroutine angeschlossen werden kann.

Das folgende Programm soll Ihnen als Programmbeispiel dienen, um Ihre eigenen Programme mit einem Stapel-Autostart versehen zu können. Da es wegen des überschriebenen Stapels nicht im Orginalbereich geschrieben werden kann, ist es ratsam, das

Programm, wie oben beschrieben, ab $1100 zu schreiben. Nachher kann die Startadresse mit dem in Kapitel 3.1.4 abgedruckten Programm auf $0100 abgeändert werden.

Unser Programm soll ein zweites Programm von Diskette nachladen und es ab $8000 starten. Als erstes muß eine kleine Routine geschrieben werden, die den Bereich von $1100 bis $11FF mit dem Wert $02 füllt. Als Startadresse haben wir $C000 (49152) gewählt.

```
C000 LDA #$02     Füllwert laden
C002 LDX #$00     Zeiger auf $00 setzen
C004 STA $1100,X  Füllwert speichern
C007 INX          Zeiger erhöhen
C008 BNE $C004    wenn kleiner $00 ($FF+1), nächster Wert
C00A RTS          Rücksprung
```

Nun folgt das eigentliche Programm:

```
1200 BRK        drei Nullbytes
1201 BRK        (Bytes sind
1202 BRK        unerheblich)
1203 LDX #$08   Gerätenummer für Floppy
1205 LDY #$01   Sekundäradresse
1207 JSR $FFBA  File-Parameter setzen
120A LDX #$1B   LOW-Byte der Filenamen-Adresse
120C LDY #$02   HIGH-Byte der Filenamen-Adresse
120E LDA #$04   Länge des Filenamens
1210 JSR $FFBD  Namensparameter setzen
1213 LDA #$00   Flagwert für LOAD (sonst VERIFY)
1215 JSR $FFD5  File laden
1218 JMP $8000  zur Startadresse springen

121B 4E 41 4D 45 00 00 00 00
```

Um das Beispiel zu vervollständigen, haben wir ein kleines Programm ab $8000 geschrieben, das die Rahmenfarbe in mehreren Schleifendurchläufen ändert.

```
8000 LDX #$00  ersten Zeiger setzen
8002 LDY #$00  zweiten Zeiger setzen
8004 INC $D020 Rahmenfarbe erhöhen
8007 INX       ersten Zeiger erhöhen
8008 BNE $8004 kleiner 255, dann weiter
800A INY       zweiten Zeiger erhöhen
800B BNE $8004 kleiner 255, dann weiter
800D RTS       Rücksprung
```

3.1.6 Autostart während des Ladens

Wie schon oben erwähnt, besitzt einer der Sprungvektoren eine besondere Eigenschaft, die sich hervorragend für eine spezielle Art des Autostarts eignet. Es handelt sich hierbei um den STOP-Vektor in $0328/$0329 (808/809), der zu einer Routine in $F6ED verzweigt. An dieser Stelle wird das STOP-Flag getestet, um den jeweiligen Programmablauf zu beenden, wenn die RUN/STOP-Taste gedrückt wurde.

Wie uns schon von den anderen Autostart-Systemen her bekannt ist, werden die einzelnen Funktionen indirekt über die Sprungvektoren angesprochen. Diese Eigenschaft haben wir uns bei dem OUTPUT-Vektor zunutze gemacht, um nach dem Laden nicht zur READY-Meldung, sondern zu unserem Programmstart zu verzweigen. Hier ist der wesentlichste Unterschied zwischen dem STOP-Vektor und den anderen Sprungvektoren festzustellen: Über den STOP-Vektor wird nicht erst nach, sondern schon während des Ladens verzweigt. Wir können also schon vorzeitig einen Programmstart durchführen.

Sicher entspricht dies nicht dem üblichen Ladevorgang, und vielleicht werden Sie sich auch fragen, welchen Sinn es hat, ein Programm zu starten, bevor es sich überhaupt vollständig im Speicher befindet, aber wir können daraus einige Vorteile ziehen.

Wenn wir bisher einen Sprungvektor umgestellt haben, um ihn auf unsere Routine zu stellen, haben wir das LOW- und HIGH-Byte verändert. Dadurch war es uns möglich, jede beliebige

Adresse anzusprechen. Da aber nach jedem Byte, das eingelesen wird, über den STOP-Vektor verzweigt wird, ist es nicht möglich, die gesamte Adresse zu ändern. Ändern wir nur das LOW-Byte, müßte zu einen Programm in $F6.. verzweigt werden, da das HIGH-Byte des Vektors $F6ED erhalten bleibt. In diesem Bereich liegt aber das Kernal-ROM, so daß dort kein eigenes Programm stehen kann.

Die einzige Möglichkeit besteht also darin, das LOW-Byte mit dem Originalwert zu überschreiben und nur das HIGH-Byte zu ändern. Zu dem Zeitpunkt, an dem der Vektor geändert wird, muß sich aber ein Teilstück des Programms bereits im Speicher befinden, zu dem wir verzweigen können. Es bleiben also nur die Adressen $00ED, $01ED oder $02ED zur Auswahl. Hier bietet sich der Bereich ab $02ED an, da die ersten beiden mit Vektoren und Zeigern belegt sind. Außerdem stehen im Bereich von $02A8 bis $02FF keine wichtigen Informationen, und es muß auch kein unbenutzter Speicherbereich zwischen unserem Programm und dem Sprungvektoren mit abgespeichert werden.

Nachdem unsere Routine gestartet wurde, kann die Originalladeroutine durch eine eigene ersetzt werden. Aus Platzmangel ist es jedoch ratsam, den Ladevorgang erst zu unterbrechen, wenn ein weiteres Programm in den Kassettenpuffer von $033C bis $03FF geladen wurde. Außerdem müssen die Sprungvektoren nach dem STOP-Vektor bis zum Ende eingeladen werden. Danach können wir weitere Programme in jeden beliebigen Speicherbereich laden.

Ein Programm, das eigentlich ein einziges langes File ist, kann sich aus dem Loader, einem BASIC-Programm ab $0801, einem Maschinensprachemonitor ab $8000 und einem Diskmonitor ab $C000 zusammensetzen. Auch können Programme unter die ROM-Bereiche geladen und zwischenzeitlich benutzt werden.

Hier nun ein Beispielprogramm, das ein Programm mit BASIC-Kopf nach $0801 lädt und dort startet. Die Sprungvektoren ab $0300 müssen natürlich unverändert mitabgespeichert werden,

damit der Rechner nicht abstürzt, wie es zum Beispiel bei der Ausgabe auf den Bildschirm oder einem Interrupt der Fall sein kann.

```
0300 8B E3 83 A4 7C A5 1A A7
0308 E4 A7 86 AE XX XX XX XX
0310 4C 48 B2 00 31 EA 66 FE
0318 47 FE 4A F3 91 F2 0E F2
0320 50 F2 33 F3 57 F1 CA F1
0328 ED 02 3E F1 2F F3 66 FE
0330 A5 F4 ED F5
```

Programm wird ab $0801 geladen:

```
0334 0B 08 C0 07
0338 9E 32 30 36 31 00 00 00
```

```
02A8 JSR $EE13      Byte vom IEC-Bus holen
02AB LDY $90        Status abfragen
02AD BNE $02C6      verzweigt, wenn Fehler (Datenende)
02AF SEI            Interrupt verhindern
02B0 TAX            Byte in X-Register retten
02B1 LDA #$34       Wert für Speicherbelegung
02B3 STA $01        RAM unter ROM einblenden
02B5 TXA            Byte wiederholen und
02B6 STA ($AE),Y    abspeichern ($AE/$AF Zieladresse)
02B8 LDA #$37       Wert für Speicherbelegung
02BA STA $01        ROM wieder einblenden
02BC CLI            Interrupt freigeben
02BD INC $AE        Zieladresse (LOW-Byte) erhöhen
02BF BNE $02A8      kein Übertrag, dann nächstes Byte
02C1 INC $AF        Zieladresse (HIGH-Byte) erhöhen
02C3 JMP $02A8      nächstes Byte holen
02C6 TYA            Statuswert in Akku schieben
02C7 AND #$FD       Bit 1 löschen (Fehler beim Lesen)
02C9 BNE $02CF      Datenende, dann $02CF
02CB STA $90        Statuswert abspeichern
02CD BEQ $02A8      unbedingter Sprung zum Anfang
```

Datenende erreicht

```
02CF JSR $F528      File schließen
02D2 LDA $AE        Zeiger (LOW) der Endadresse
02D4 STA $2D        in Zeiger für BASIC-Programmende
02D6 LDA $AF        Zeiger (HIGH)
02D8 STA $2E        in BASIC-Programmende
02DA LDA #$ED       LOW-Byte STOP-Vektor
02DC STA $0328      abspeichern
02DF LDA #$F6       HIGH-Byte STOP-Vektor
02E1 STA $0329      abspeichern
02E4 LDA #$00       Code-Wert für Schwarz
02E6 STA $D020      Rahmenfarbe setzen
02E9 JMP $080D      Programmstart
02EC NOP            No Operation
```

Einsprung über STOP-Vektor ($0328/$0329)
Der Ladevorgang wird erst unterbrochen, wenn alle Sprungvektoren eingelesen sind.

```
02ED LDA $AE        Zieladresse (LOW-Byte) laden
02EF CMP #$34       mit $0334 vergleichen
02F1 BEQ $02F4      Adresse erreicht, dann weiter
02F3 RTS            sonst weiterladen
02F4 LDA #$01       neue Zieladresse (LOW-Byte)
02F6 STA $AE        abspeichern
02F8 LDA #$08       neue Zieladresse (HIGH-Byte)
02FA STA $AF        abspeichern
02FC BNE $02A8      unbedingter Sprung zu $02A8
```

```
0340 (080D) LDX #$00    1. Zeiger setzen
0342 (080F) LDY #$00    2. Zeiger setzen
0344 (0812) INC $D020   Rahmenfarbe erhöhen
0347 (0815) INX         1. Zeiger vermindern
0348 (0816) BNE $0344   kleiner 255, dann weiter
034A (0818) INY         2. Zeiger vermindern
```

```
034B (0819) BNE $0344   kleiner 255, dann weiter
034D (081B) BRK         Rücksprung
```

Ab der Adresse $0340 liegt ein kleines Programm, das nach dem Autostart ab $080D liegen wird. Die BASIC-Startzeile mit dem SYS-Befehl ist oben mit den Sprungvektoren ab $0334 abgedruckt. Anstelle unseres kleinen Beispielprogramms können Sie natürlich ein beliebiges anderes Programm einsetzen, das ab $080D gestartet wird. Dazu steht Ihnen der gesamte Speicher zur Verfügung.

Da Sie jetzt in der Lage sind, die Programme in beliebige Speicherbereiche zu laden, könnten Sie eventuell wie folgt fortfahren: Setzen Sie unmittelbar hinter das Programm, das ab $080D beginnt, ein weiteres, das Sie zum Beispiel ab $F000 laden. Nach dem Laden können Sie dann mit

```
LDA #$34
STA $01
```

den gesamten ROM-Bereich auf das darunterliegende RAM umblenden. Hier sind Ihrer Kreativität keine Grenzen gesetzt.

3.1.7 Autostart über Interrupt

Nach dem unter 3.1.6 beschriebenen Verfahren kann ein Autostart während des Ladevorgangs auch über den Interrupt-Vektor in $0314/$0315 (788/789) durchgeführt werden. Der Interrupt-Vektor wird alle 60stel Sekunde benutzt, um verschiedene Funktionen durchzuführen. Dazu gehören unter anderem die Cursorfunktion, die Abfrage einzelner Tasten und die Datasettensteuerung.

Wenn wir den IRQ-Vektor benutzen, kann das Prinzip des STOP-Vektors beibehalten werden. Allerdings stellt sich hier das Problem, den Interrupt genau zu berechnen. Im Gegensatz zum STOP-Vektor können wir hier nicht genau sagen, nach welchem eingeladenen Byte der Vektor wieder benutzt wird. Die Routine ab $02ED, die solange in die Originalladeroutine zurückspringt,

bis ein bestimmtes Byte geladen wurde, kann hier nicht mehr verwendet werden. Stattdessen müssen wir sofort unsere eigene Laderoutine verwenden und mit dieser bis zu einem bestimmten Byte laden. Achten Sie darauf, daß zu Beginn Ihrer Routine ein SEI-Befehl steht, jeder weitere Interrupt also unterdrückt wird. Ohne diesen Befehl würde das Programm beim nächsten Interrupt erneut einspringen, was zu einem totalen Chaos und einem sicheren Absturz des Rechners führen würde.

3.1.8 Autostart über die CIAs

Neben den diversen Möglichkeiten, einen Autostart über den Stapel oder die Sprungvektoren durchzuführen, kann man zu diesem Zweck auch die CIAs verwenden. Dabei wird zu einem bestimmten Zeitpunkt ein NMI, also ein nicht maskierbarer Interrupt, von einem Timer der CIAs ausgelöst.

An dieser Stelle taucht der erste Unterschied zu den bisher besprochen Autostartmethoden auf, der uns gleichzeitig auch vor ein Problem stellt. Wenn von den Timern ein NMI ausgelöst wird, so wird über den NMI-Vektor in $0318/$0319 (792/793) zur Adresse $FE47 verzweigt. Zwar kann dieser Vektor in gewohnter Manier auf unsere Routine gestellt werden, dieses muß jedoch vor dem Ladevorgang erfolgen. Daraus folgt, daß ein automatischer Programmstart nur von einem Ladeprogramm durchgeführt werden kann, was die ganze Sache ziemlich absurd und sinnlos erscheinen läßt.

Doch auch wenn die CIAs nicht für einen richtigen Autostart zu gebrauchen sind, kann hier ein wirksamer Programmschutz installiert werden. Das nun folgende Beispiel soll verdeutlichen, wie die Timer gestellt werden müssen, um während des Ladens einen NMI an einer bestimmten Stelle auszulösen.

Dazu muß der NMI-Vektor in $0318/$0319 (792/793) zuerst auf $2000 gestellt werden. Das kann mittels POKE, eines kleinen Maschinenspracheprogramms oder durch Überschreiben des Vektors während des Ladens geschehen. Das eigentliche Programm muß ab $DD06 geladen werden, damit die Werte direkt

in den Speicherzellen für den Timer stehen und die anderen Funktionen der CIAs nicht beeinträchtigt werden.

In den Speicherzellen $DD06 und $DD07 steht im LOW- und HIGH-Byte-Format der Wert, von dem aus der Timer B auf Null herunterzählt. Dieser Wert steht hier auf $0001, so daß nach dem Starten des Timers B durch das Verändern des Wertes in $DD0F ein NMI ausgelöst wird. Die nächsten vier Bytes sind für die Echtzeituhr zuständig und interessieren an dieser Stelle nicht. Das folgende Byte stellt das "Serial Data"-Register dar und ist für uns auch uninteressant.

In $DD0D, dem "Interrupt Control"-Register, müssen die Bits 1 und 7 gesetzt werden. Bit 1 = 1 zeigt an, daß ein Unterlauf von Timer B stattgefunden hat. Wenn ein Bit aus diesem Register gesetzt ist, so muß Bit 7 auch gesetzt sein, um das anzuzeigen.

Der Wert $80 in $DD0E heißt, daß Bit 7 = 1 ist, was die Taktfrequenz des Echtzeituhr-Triggers auf 50 Hz stellt. Aber auch dieses Register beeinflußt den NMI von Timer B nicht. Interessant wird es wieder in Register 15, also in $DD0F. Hier muß Bit 0 gesetzt werden, um den Start des Timers B anzuzeigen. Bit 5 und 6 müssen gelöscht sein, damit Timer B die Systemtakte zählt.

Die Inhalte der Speicherzellen $DD06 bis $DD0F könnten also folgendermaßen aussehen:

```
DD00 XX XX XX XX XX XX 01 00
DD08 00 00 00 91 00 82 80 19
```

Diese Bytes, die ein beliebiges Programm darstellen sollen, stehen nach dem Laden ab $3000.

```
DD10 01 02 03 04 05 06 07 08
DD18 09 10 11 12 13 14 15 16
DD20 17 18 19 20 21 22 23 24
```

Wenn Sie diese Datenfolge auf einer Diskette ablegen wollen, so speichern Sie bitte die Bytes zuerst aus einem anderen Speicher-

bereich ab, zum Beispiel $1D00, und ändern nachträglich die Startadresse mit dem Programm aus 3.1.4 auf $DD00 um.

Das hier abgedruckte Programm wird nach dem NMI gestartet und setzt die Zieladresse der nächsten Bytes, die geladen werden, auf $3000. Das Programm kann mit einem RTI (Return from Interrupt) beendet werden, da es durch einen Interrupt aufgerufen wurde.

```
2000 SEI          maskierbaren Interrupt verhindern
2001 LDX #$00     LOW-Byte der neuen Zieladresse
2003 LDY #$30     HIGH-Byte der neuen Zieladresse
2005 STX $AE      LOW-Byte abspeichern
2007 STY $AF      HIGH-Byte abspeichern
2009 CLI          Interrupt wieder freigeben
200A RTI          Rücksprung vom Interrupt (NMI)
```

Wärend des Ladens sind keine Veränderungen sichtbar. Trotzdem stehen die Bytes jetzt ab der Adresse $3000.

```
3000 01 02 03 04 05 06 07 08
3008 09 10 11 12 13 14 15 16
3010 17 18 19 20 21 22 23 24
```

3.1.9 Autostart mit Adreßverschiebung

Wie in Kapitel 3.1.6 beschrieben, ist es möglich, einen Autostart während des Ladens durchzuführen. Mit diesem Autostart konnten wir durch das Verändern der Speicherzellen $AE/$AF (174/175) die Zieladresse eines Programmes umstellen. Da die Adresse des jeweils zu ladenden Bytes in diesen Speicherzellen (im LOW- und HIGH-Byte-Format) abgelegt ist, kann sie auf eine beliebige Endadresse abgeändert werden, ab der das Programm dann weitergeladen wird.

Diese Zieladresse muß aber nicht unbedingt vom Programm aus verändert werden. Die Speicherzellen $AE/$AF können während des Ladens auch direkt überschrieben werden, wenn das Origi-

nalprogramm vor dieser Adresse zu laden beginnt. Wird nun das LOW-Byte der Zieladresse verändert, ergibt das nicht viel Sinn, da das HIGH-Byte immer noch auf $00 steht (Zieladresse $00AE) und so nur innerhalb der Zeropage weitergeladen werden kann. Es ist also sinnvoll, das LOW-Byte mit dem Originalwert zu überschreiben und nur das HIGH-Byte zu ändern.

Da wir am Ende des Ladeprogrammes wieder einen Autostart durchführen wollen, also in den Bereich zwischen $0100 und $033C weitere Bytes schreiben müssen, und da trotzdem ein Programmteil zwischen $0800 und $FFFF liegen soll, müssen wir uns hier eines kleinen Tricks bedienen. Wir stellen den Zeiger in $AE/$AF so um, daß zum Beispiel in den Bereich ab $FE00 plus dem LOW-Byte geladen wird. In diesem Bereich liegt zwar momentan ROM-Bereich, was uns aber nicht weiter stören soll, da beim Laden die Bytes in den darunterliegenden RAM-Bereich geschrieben werden.

Es wird nun also ein beliebiges Programm in den Bereich ab $FE00 geschrieben, das später dort gestartet werden kann. Das funktioniert natürlich nur, wenn der Bereich zuvor mit

```
LDA #$35
STA $01
```

auf den darunterliegenden RAM-Bereich umgeschaltet wurde.

Nun muß das Programm bis $FFFF weitergeladen werden. Wenn Ihr Programm nicht so lang ist, kann der restliche Bereich auch mit Nullbytes aufgefüllt werden. Für den Fall, daß Ihnen für das Programm nicht genug Platz zur Verfügung steht, kann das HIGH-Byte des Zeigers in $AE/$AF auf eine beliebige andere Zieladresse gestellt werden.

Nachdem ein Byte in $FFFF abgelegt wurde, die letzte Adresse des Speichers also erreicht wurde, wird das nächste Byte am Anfang des Speichers abgelegt. Es wird also ab $0000 weitergeladen. Von hier an kann nun normal weitergeladen werden, wenn die Zeropage-Adressen mit Originalwerten überschrieben werden. Der Inhalt der Zeropage kann zuvor ausgelesen und

später mit diesen Werten wieder überschrieben werden. Auf die Adressen $AE/$AF ist hier wieder besonders zu achten.

Am Ende kann ein beliebiger Autostart, zum Beispiel mit Hilfe des Stapels, durchgeführt werden. Danach muß der ROM-Bereich, in dem unser Programm liegt, auf den darunterliegenden RAM-Bereich umgeschaltet werden, und unser Programm kann gestartet werden. Doch auch hier können Sie noch weitere, eigene Ideen mit einbauen, um Ihr Programm für andere so "unverständlich" wie möglich zu gestalten.

3.2 Prüfsummen und Selbstzerstörung

Das Bilden einer Prüfsumme von einem Maschinenprogramm eignet sich hervorragend als Änderungsschutz. Falls man sein Maschinenprogramm vor einem Fremdeingriff schützen möchte, bildet man einfach eine Prüfsumme von seinem Maschinenprogramm und fragt diese irgendwann mitten im Programmablauf ab.

Es gibt einige Methoden, Prüfsummen zu bilden. Wir haben hier ein Prüfsummen-Programm für Sie vorbereitet:

```
2000 LDY #$00      ersten Zähler setzen
2002 LDX #$10      zweiten Zähler setzen
2004 LDA #$00      LOW-Byte der Anfangsadresse laden
2006 STA $26       und speichern
2008 LDA #$10      HIGH-Byte der Anfangsadresse laden
200A STA $27       und speichern
200C LDA ($26),Y   Wert laden
200E CLC           und mit dem
200F ADC $02       Wert aus Speicherzelle $02 addieren
2011 STA $02       und wieder in $02 speichern
2013 INY           Zähler erhöhen
2014 BNE $200C     noch keine 255 Bytes, dann nächstes Byte
2016 INC $27       HIGH-Byte erhöhen
2018 DEX           Zähler vermindern
2019 BNE $200C     falls Zähler null, dann weiter
```

```
201B LDA $02      berechneten Prüfsummenwert laden
201D CMP #$36     und mit vorgegeben Wert vergleichen
201F BEQ $2029    falls Werte gleich, dann Rücksprung
2021 LDX #$00     Zähler auf Null setzen
2023 INC $D020    Rahmenfarbe erhöhen
2026 INX          Zähler erhöhen
2027 BNE $2023    falls 255 mal erhöht, dann Rücksprung
2029 RTS          Rücksprung
```

Das Programm bildet eine Prüfsumme von dem Speicherbereich $1000 - $1FFF. Die Bereiche lassen sich begrenzen, indem man den Zähler in Zeile $2002 entsprechend setzt. Auch der zu prüfende Speicherbereich läßt sich angeben, indem man das LOW- und HIGH-Byte in Zeile $2004 und $2008 entsprechend umstellt. Anschließend wird ein Byte aus dem vorgegebenen Bereich geladen und zu dem Wert aus der Speicherzelle $02 addiert. Dieser Wert ist beim ersten Durchlauf der Schleife gleich Null. Beim zweiten Durchlauf steht dort der Wert der ersten geladenen Speicherzelle. Beim dritten Durchlauf wird der geholte Wert mit dem in $02 addiert. Bei einem Überlauf, also wenn die Summe 255 überschreitet, wird wieder von Null an weiteraddiert.

Nach der Prüfsummenbildung wird der berechnete Wert aus der Speicherzelle $02 geladen und mit einem vorgegebenen Wert verglichen. Hier muß natürlich auch die richtige Prüfsumme von dem eigenen Programm stehen. Diese läßt sich auch mit dem gleichen Programm berechnen. Nach der Prüfsummenberechnung braucht man nur den Wert aus dem Akku zu lesen und in die Zeile $201D einzusetzen.

Falls nun eine Änderung an dem geprüften Programm vorliegt, und die Prüfsumme nicht stimmt, wird bei unserem Beispielprogramm nur die Rahmenfarbe erhöht. Anschließend folgt der Rücksprung. Falls die Prüfsumme stimmt, also wenn keine Änderungen am Programm vorliegen, wird direkt zum Rücksprung verzweigt.

In der Praxis sollte an dieser Stelle statt einer Rahmenfarberhöhung eine Selbstzerstörung stattfinden. Das heißt, daß einfach wichtige Programmteile zerstört werden und folglich das Programm nicht mehr lauffähig ist. Ein Lösch- oder Zerstörungsprogramm könnte folgendermaßen aussehen:

```
033C SEI          Interrupt verhindern
033D LDA #$30     I/O-Bereich und Betriebssystem
033F STA $01      ausblenden
0341 LDX #$FB     ersten Zähler setzen
0343 LDY #$00     zweiten Zähler setzen
0345 LDA #$00     LOW-Byte der Anfangsadresse laden
0347 STA $26      und speichern
0349 LDA #$04     HIGH-Byte der Anfangsadresse laden
034B STA $27      und speichern
034D LDA #$00     Wert laden
034F STA ($26),Y  und speichern
0351 INY          Zähler erhöhen
0352 BNE $034F    falls 256 Bytes, dann
0354 INC $27      HIGH-Byte der Anfangsadresse erhöhen
0356 DEX          Zähler erniedrigen
0357 BNE $034F    falls Zähler Null dann
0359 LDA #$37     Betriebssystem wieder
035B STA $01      einblenden
035D CLI          Interrupt verhindern
035E JMP $FCE2    zum RESET springen
```

Dieses Löschprogramm füllt den Speicherbereich von $0400 bis $FFFF mit Nullen und springt anschließend zum RESET, wodurch das Löschprogramm wiederum zerstört wird. Das Betriebssystem und der I/O-Bereich müssen ausgeblendet werden, weil beim Überschreiben des I/O-Bereichs der Computer abstürzen würde. Nach dem Auffüllen wird noch die RESET-Routine des Betriebssystems angesprungen, um die Löschroutine zu zerstören. Auf diese Weise kann man auch vor einem Programmstart unbenutzte Bereiche füllen, um irgendwelche Fremdprogramme zu zerstören.

3.3 Codierung von Programmen

Neben den Illegal-Opcodes, die wir in Kapitel 3.4 beschrieben haben, gibt es noch eine andere Methode, eigene Programme und speziell Kopierschutzabfragen vor Fremdeingriffen zu schützen. In dem nun folgenden Kapitel soll die Codierung von Programmen erklärt werden.

3.3.1 Verwendung der EXOR-Verknüpfung

Zuerst sollte der Begriff der Codierung allgemein erklärt werden, um eventuellen Mißverständnissen vorzubeugen.

Codieren bedeutet, z.B. einen Text nach einen bestimmten System zu verschlüsseln. Der codierte Text kann nun nicht mehr gelesen werden, ohne daß man ihn zuvor mit demselben System wieder zu decodiert. Da ein Programmlisting, ähnlich wie ein Text auch, gelesen werden kann, müssen wir ein System finden, das die einzelnen Bytes eines Programms nach einer bestimmten Logik verändert, so wie es auch bei den einzelnen Buchstaben eines Textes der Fall ist.

Dann muß das Programm in codierter Form mit einer zusätzlichen Decodierroutine abgespeichert werden. Diese Routine muß vor dem Programmablauf gestartet werden, um die Bytes wieder in ihren Originalzustand zu versetzen. Systeme, mit denen eine Byte-Folge verändert werden kann, gibt es sicherlich recht viele, und einige davon werden wir Ihnen in diesen Kapitel vorstellen. Am einfachsten ist dieses Ziel jedoch mit der EXOR-Verknüpfung zu erreichen. EXOR ist eine logische Verknüpfung aus der Booleschen Algebra, die auf andere Verknüpfungen aufbaut. Es würde zu weit führen, an dieser Stelle die gesamte Grundstruktur dieser Algebra zu erklären, und deshalb beschränken wir uns hier auf die EXOR-Verknüpfung und den entsprechenden Assemblerbefehl EOR.

Wie Sie sicherlich wissen, bestehen ein Assemblerbefehl und die dazugehörigen Parameter aus einem bzw. mehreren Bytes, die

sich jeweils aus acht Bits zusammensetzen. Verknüpft man ein Programm-Byte mit einem beliebigen anderen Byte, werden die einzelnen Bits der Bytes miteinander verknüpft. Man betrachtet also zuerst die beiden ersten Bits, dann die zweiten beiden Bits und so weiter. Sind alle acht Bits miteinander verknüpft worden, ergibt sich daraus eine völlig neue Bitfolge, also ein neues Byte.

Bei der EXOR-Verknüpfung ist das Ergebnisbit immer dann gleich 1, wenn beide zu verknüpfenden Bits gleich lang sind. Sind die Bits gleich, so ist das Ergebnisbit gleich 0. Hier die dazugehörige Tabelle:

```
EXOR

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

Anhand eines Beispiels wollen wir zeigen, wie zwei Bytes miteinander verknüpft werden:

```
     Byte 1 = 10000101 = $85 = 133
EXOR Byte 2 = 00101010 = $2A =  42
-------------------------------------

  Ergebnis  : 10101111 = $AF = 175
```

Eine spezielle Eigenart dieser Verknüpfung, die vor allem bei der Codierung weiterhilft, besteht darin, daß ein Byte A, wird es zweimal mit einem Byte B EXOR-verknüpft, wieder das ursprüngliche Byte A ergibt. Das können wir an dem obigen Beispiel recht einfach nachvollziehen:

```
     Byte A = 10000101 = $85 = 133
EXOR Byte B = 00101010 = $2A =  42
-------------------------------------

  Ergebnis  : 10101111 = $AF = 175
EXOR Byte B = 00101010 = $2A =  42
-------------------------------------

  Ergebnis  : 10000101 = $85 = 133
```

Mit dieser Verknüpfung besitzen wir ein fast optimales System zur Programmcodierung. Eine Routine, die das Hauptprogramm codiert und auch als Decodierroutine mit abgespeichert werden kann, könnte so aussehen:

```
1000 LDX #$00      Zeiger auf $00 setzen
1002 LDA $2000,X   Byte holen und
1005 EOR #$F4      (de)codieren
1007 STA $2000,X   Byte zurückschreiben
100A INX           Zeiger erhöhen
100B BNE $1002     zum Schleifenanfang, wenn kleiner 256
100D RTS           Rücksprung
```

Nach dem Abarbeiten steht das Hauptprogramm codiert im Speicherbereich von $2000 bis $20FF und kann so abgespeichert werden. Um die Codierroutine auch zum Decodieren benutzen zu können, muß der RTS-Befehl in $100D durch einen JMP-Befehl zum Programmstart ersetzt werden. Außerdem sollte die Routine unmittelbar vor oder hinter dem Hauptprogramm stehen, um den dazwischenliegenden Speicherbereich nicht mit absichern zu müssen.

Um Programme zu codieren, die länger als 256 Bytes sind, muß in der Routine noch ein zweiter Zähler benutzt werden, der die Anzahl der 256-Byte-Blöcke zählt. Außerdem muß nach jedem Block das HIGH-Byte der Zieladressen erhöht werden.

```
2027 LDX #$00      ersten Zeiger setzen
2029 LDY #$00      zweiten Zeiger setzen
202B LDA $5000,X   Programm-Byte holen
202E EOR #$F4      Byte verknüpfen
2030 STA $5000,X   Programm-Byte wieder abspeichern
2033 INX           ersten Zeiger erhöhen
2034 BNE $202B     zum Schleifenanfang, wenn kleiner 256
2036 INC $202D     HIGH-Byte des Programmzeigers erhöhen
2039 INC $2032     HIGH-Byte der Zieladresse erhöhen
203C INY           zweiten Zeiger erhöhen
```

```
203D CPY #$10     schon $10 Blöcke (de)codiert?
203F BNE $202B    wenn nicht, dann zum Schleifenanfang
2042 RTS          Rücksprung
```

3.3.2 Codierung mit verstecktem Einsprung

Das in Kapitel 3.3.1 beschriebene System zeichnet sich durch seine Einfachheit aus und ist aus diesem Grunde besonders für Leute geeignet, die noch keine große Erfahrung haben. Der Codierschutz ist aber recht einfach zu entfernen, indem man den Programmeinsprung am Ende der Decodierroutine durch ein BRK oder ein RESET (JMP $FCE2) ersetzt. Dann kann die Decodierroutine gestartet werden, ohne daß das Hauptprogramm abläuft. Da der Einsprung durch den JMP-Befehl auch bekannt ist, kann das nun decodierte Programm leicht nachverfolgt und ein eventuell vorhandener Kopierschutz entfernt werden. Wenn der Einsprung aber unbekannt ist, läßt sich das Programm nicht mehr so leicht nachverfolgen.

Als Beispiel haben wir eine Routine erstellt, die das Programm "von hinten her" decodiert. Betrachtet man die Decodierroutine, sieht man nur eine Schleife, an deren Ende ein BRK-Befehl steht, welcher den Programmfluß nach der Routine stoppen müßte. In Wirklichkeit verzeigt der BNE-Befehl in $100B aber nur solange zu $1002, bis die Speicherzelle $100D decodiert und damit verändert wird. Der BNE-Befehl verzweigt aber intern gar nicht zu der Adresse $1002, sondern nur eine bestimmte Anzahl von Bytes nach vorne oder hinten. Im Speicher steht also nur der Operationscode für BNE und danach ein Byte, das den Einsprung bestimmt. Ist dises Byte zum Beispiel 10, so wird 10 Bytes nach vorne verzweigt. Alle Werte, die größer als 127 sind, veranlassen einen Sprung in die andere Richtung. Das heißt, daß die Bits 0 bis 7 die Schrittweite angeben, die also maximal 127 Byte betragen kann, und daß Bit 7 die Richtung bestimmt.

Auf unser Programmbeispiel bezogen bedeutet das, daß wir den Wert, mit dem codiert wird, so wählen müssen, daß aus dem BNE $1002 nachher ein BNE $100E wird. In diesem Fall wäre das der Wert $F4.

Unser Beispiel haben wir bewußt einfach gestaltet, so daß es leicht nachzuvollziehen ist. In der Praxis können Sie das Programm anders gestalten, indem Sie zum Beispiel dem BRK-Befehl durch einen völlig unsinnigen JMP-Befehl ersetzen oder eine ganze Routine schreiben, die jeden, der das Prinzip nicht kennt, völlig verwirrt.

Hier nun das Programmbeispiel mit einem kleinen Hauptprogramm und der dazugehörigen Codierroutine:

```
1000 LDX #$FF      Zeiger auf $FF setzen
1002 LDA $1000,X   codiertes Byte holen
1005 EOR #$F4      mit Wert $F4 decodieren
1007 STA $1000,X   Byte zurückschreiben
100A DEX           Zeiger um 1 vermindern
100B BNE $1002     unbedingter Sprung zum Schleifen-
                   anfang, da $00 nie erreicht wird
100D BRK           scheinbares Programmende
```

Einsprung nach Abändern des BNE-Wert durch EOR #$F4:

```
100E LDX #$00      ersten Zeiger setzen
1010 LDY #$00      zweiten Zeiger setzen
1012 INC $D020     Rahmenfarbe erhöhen
1015 INX           ersten Zeiger vermindern
1016 BNE $1012     zum Schleifenanfang
1018 INY           zweiten Zeiger erhöhen
1019 BNE $1012     zum Schleifenanfang
101B BRK           Programmende
```

Um das oben abgedruckte Programm zu codieren, brauchen wir ein Programm, das folgendermaßen aussehen könnte. Der Zeiger wird dabei auf $0E gesetzt, weil das erste zu codierende Byte, der LDX-Befehl, in $1000 + $0E = $100E steht.

```
2000 LDX #$0E      Zeiger auf $0E setzen
2002 LDA $1000,X   erstes Byte holen
2005 EOR #$F4      Byte codieren
2007 STA $1000,X   Byte zurückschreiben
200A INX           Zeiger erhöhen
200B BNE $2002     zum Schleifenanfang
200D RTS           Rücksprung
```

Das Programm besteht nun aus einem codierten und einem uncodierten Teil und sieht wie folgt aus:

```
1000 LDX #$FF      Zeiger setzen
1002 LDA $1000,X   Byte holen und
1005 EOR #$F4      decodieren
1007 STA $1000,X   Byte zurückschreiben
100A DEX           Zeiger vermindern
100B BNE $1002     zum Schleifenanfang
100D BRK           scheinbares Programmende
100E LSR $F4,X     codiertes Programm
1010 ???                 .
1011 ???                 .
1012 ???                 .
1013 ???                 .
1014 BIT $1C             .
1016 BIT $0E             .
1018 ???                 .
1019 BIT $03             .
101B ???                 .
101C ASL $1E1E,X         .
101F ASL $1E1E,X   Ende des codierten Programms
```

3.3.3 Codierung mit Timer

An den Schluß dieses Kapitels haben wir einen Unterpunkt gesetzt, der sich mit einem komplexeren Codierungssystem beschäftigt. In den bislang beschriebenen Beispielen wurde jedes Byte eines Programmes immer mit dem gleichen Code verschlüsselt. Hier haben wir nun ein System gewählt, bei dem sich

der Codewert andauernd ändert. Zu diesem Zweck lassen wir einen Timer der CIAs die Taktzyklen zählen und benutzen den Zählerwert als als Codewert.

Bei dieser Codiermethode muß man darauf achten, daß sowohl die Codier- als auch die Decodierroutine sich zeitlich nicht voneinander unterscheiden. Das nun folgende Programm soll demonstrieren, welche Schwierigkeiten dabei entstehen. Es läßt dazu den Timer B eine bestimmte Zeit lang laufen und legt dann den ausgelesenen Wert im LOW-/HIGH-Byte-Format im Speicher ab.

```
2000 SEI          Interrupt verhindern
2001 LDA #$00     Wert laden und
2003 STA $DD0F    Timer stoppen
2006 LDA #$FF     LOW-Byte des Wertes
2008 STA $DD06    in den Timer schreiben
200B LDA #$FF     HIGH-Byte des Wertes
200D STA $DD07    in Timer schreiben
2010 LDA #$99     Wert für "Timer im Continue-Modus
2012 STA $DD0F    starten" in Register schreiben
2015 LDX #$00     Zeiger auf $00 setzen (Warteschleife)
2017 INX          Zeiger erhöhen
2018 BNE $2017    zum Schleifenanfang, wenn kleiner 256
201A LDA $DD06    LOW-Byte des Timers laden und
201D STA $4000    Wert in $4000 abspeichern
2020 LDA $DD07    HIGH-Byte laden und
2023 STA $4001    Wert in 4001 abspeichern
2026 LDA #$00     Wert laden und damit
2028 STA $DD0F    Timer stoppen
202B CLI          Interrupt freigeben
202C RTS          Rücksprung
```

Das BASIC-Programm soll die oben abgedruckte Routine starten und den Timerwert im LOW- und HIGH-Byte-Format ausgeben. In Zeile 30 wird das Programm wiederholt.

```
10 SYS2*4096
20 PRINT PEEK(4*4096),PEEK(4*4096+1)
30 GOTO10
```

Wie Sie beim Programmablauf bestimmt festgestellt haben, ändern sich die Werte jedesmal. Das läßt sich dadurch erklären, daß der Timer, im Gegensatz zum Programm, kontinuierlich läuft. Zwar haben wir im Programm durch den SEI-Befehl alle maskierbaren Interrupts ausgeschaltet, doch der Video-Controller unterbricht das Programm immer noch. Um dieses zu verhindern, müssen wir vorher den Bildschirm ausschalten.

Da der Bildschirm nicht an jeder beliebigen Stelle, sondern erst am Ende des Rasterstrahldurchlaufs ausgeschaltet werden kann, muß eine Warteschleife eingefügt werden. Diese liegt in dem folgenden Programm von $2009 bis $2011.

```
2000 SEI          Interrupt verhindern
2001 LDA $D011    Wert aus Steuerregister 1 laden
2004 AND #$EF     Bit 4 löschen (Bildschirm ausschalten)
2006 STA $D011    Wert zurückschreiben
2009 LDX #$00     erster Zeiger für Warteschleife
200B LDY #$F0     zweiter Zeiger für Warteschleife
200D INX          ersten Zeiger erhöhen
200E BNE $200D    zum Schleifenanfang, wenn kleiner 256
2010 INY          zweiten Zeiger erhöhen
2011 BNE $200D    zum Schleifenanfang, wenn kleiner 256
2013 LDA #$00     Wert laden und
2015 STA $DD0F    Timer stoppen
2018 LDA #$FF     LOW-Byte für
201A STA $DD06    Timer setzen
201D LDA #$FF     HIGH-Byte für
201F STA $DD07    Timer setzen
2022 LDA #$99     Timer im Continue-Modus
2024 STA $DD0F    starten
2027 LDX #$00     Zeiger setzen (Warteschleife)
2029 INX          Zeiger erhöhen
202A BNE $2029    zum Schleifenanfang, wenn kleiner 256
202C LDA #$00     Timer
202E STA $DD0F    stoppen
```

```
2031 LDA $D011   Bildschirm
2034 ORA #$10    wieder
2036 STA $D011   einschalten (Bit 4 setzen)
2039 LDA $DD06   LOW-Byte des Timerwertes
203C STA $4000   in $4000 speichern
203F LDA $DD07   HIGH-Byte
2042 STA $4001   in $4001 speichern
2045 CLI         Interrupt wieder freigeben
2046 RTS         Rücksprung
```

Auch dieses Programm kann wieder mit dem BASIC-Programm gestartet werden. Nun wird der Bildschirm kurzzeitig ausgeschaltet, was sich durch ein Flimmern bemerkbar macht. Die ausgelesenen Timerwerte sind jetzt immer identisch.

Nun kommen wir zu der eigentlichen Codierroutine, die die einzelnen Bytes des Programmes mit den Werten aus dem Timer codiert. Da der Timer kontinuierlich abläuft, entsteht für jedes codierte Byte ein anderer Wert. Die einzelnen Programm-Bytes werden dadurch EXOR-verknüpft. Daraus ergibt sich, daß das folgende Programm auch als Decodierprogramm benutzt werden kann, wenn die Anfangswerte für den Timer unverändert bleiben.

```
2000 SEI         Interrupt verhindern
2001 LDA $D011   Bildschirm
2004 AND #$EF    ausschalten
2006 STA $D011   (Bit 4 löschen)
2009 LDX #$00    ersten Zeiger laden
200B LDY #$F0    zweiten Zeiger laden
200D INX         ersten Zeiger erhöhen
200E BNE $200D   zum Schleifenanfang, wenn kleiner 256
2010 INY         zweiten Zeiger erhöhen
2011 BNE $200D   zum Schleifenanfang, wenn kleiner 256
2013 LDA #$00    Timer
2015 STA $DD0F   stoppen
2018 LDA #$FF    LOW-Byte des
201A STA $DD06   Timers setzen
201D LDA #$FF    HIGH-Byte des
```

```
201F STA $DD07    Timers setzen
2022 LDA #$99     Timer im Continue-Modus
2024 STA $DD0F    starten
2027 LDX #$00     ersten Zeiger setzen
2029 LDY #$F0     zweiten Zeiger setzen
202B LDA $5000,X  Programm-Byte holen
202E EOR $DD06    mit LOW-Byte des Timers verknüpfen
2031 STA $5000,X  Programm-Byte wieder abspeichern
2034 INX          ersten Zeiger erhöhen
2035 BNE $202B    zum Schleifenanfang, wenn kleiner 256
2037 INC $202D    HIGH-Byte des Programmzeigers erhöhen
203A INC $2033    HIGH-Byte der Zieladresse erhöhen
203D INY          zweiten Zeiger erhöhen
203E BNE $202B    zum Schleifenanfang, wenn kleiner 256
2040 LDA #$50     ursprüngliches HIGH-Byte der
2042 STA $202D    Programm- und
2045 STA $2033    Zieladresse wieder herstellen
2048 LDA #$00     Timer
204A STA $DD0F    stoppen
204D LDA $D011    Bildschirm
2050 ORA #$10     wieder einschalten
2052 STA $D011    (Bit 4 setzen)
2055 CLI          Interrupt wieder freigeben
2056 RTS          Rücksprung
```

Die folgenden Bytes sollen ein beliebiges Programm darstellen, das codiert werden soll:

```
5000 55 55 55 55 55 55 55 55
5008 55 55 55 55 55 55 55 55
5010 55 55 55 55 55 55 55 55
5018 55 55 55 55 55 55 55 55
5020 55 55 55 55 55 55 55 55
5028 55 55 55 55 55 55 55 55
5030 55 55 55 55 55 55 55 55
```

Nach dem Ablauf der Codierroutine sieht der Speicherbereich folgendermaßen aus:

```
5008 33 01 17 65 4B 59 AF BD
5010 83 91 E7 F5 DB 29 3F 0D
5018 13 61 77 45 AB B9 8F 9D
5020 E3 F1 C7 D5 3B 09 1F 6D
5028 73 41 57 A5 8B 99 EF FD
5030 C3 D1 27 35 1B 69 7F 4D
```

Nach erneutem Durchlauf des Programmes sieht der Speicherbereich wieder wie oben aus.

Das Decodierprogramm haben wir hier als BASIC-Loader abgedruckt, für den Fall, daß Sie keinen Assembler besitzen oder daß Ihnen das Abtippen des Assemblercodes zu mühselig erscheint.

```
10 FOR X=0 TO 86
20 READ A:POKE2*4096+X,A
30 S=S+A
40 NEXT X
50 IF S<>11452 THEN PRINT"FEHLER IN DATAS":STOP
60 REM SYS2*4096
100 DATA120,173,17,208,41,239,141,17,208,162,0,160,240,232,208,253,200,2
08
110 DATA250,169,0,141,15,221,169,255,141,6,221,169,255,141,7,221,169,153
120 DATA141,15,221,162,0,160,240,189,0,80,77,6,221,157,0,80,232,208,244,
238
130 DATA45,32,238,51,32,200,208,235,169,80,141,45,32,141,51,32,169,0,141
140 DATA15,221,173,17,208,9,16,141,17,208,88,96
```

3.3.4 Einzelschritt-Decodierung

Bei vielen Programmschutz-Anwendungen stellt sich die Frage: Wie verhindere ich, daß ein "Knacker" mitten im Programmlauf seinen RESET-Schalter betätigt und sich dann in aller Ruhe das Schutzprogramm ansieht? Ein 'CBM80' (siehe Kapitel 2.3) im

Speicher ist nicht zuverlässig genug, und nicht immer hat man für Programme Speicherbereiche frei, die bei einem RESET gelöscht werden (siehe Kapitel 7.1).

Es gibt allerdings noch eine andere Methode, sich gegen unerwünschte Blicke zu wehren, nämlich indem man sein Programm codiert. Ein Programm muß natürlich vor seinem Ablauf erst wieder decodiert werden, aber es reicht ja, immer nur den nächsten Befehl, der ausgeführt werden soll, zu entschlüsseln. Der vorhergehende Befehl wird nach seinem Ablauf wieder codiert. Dadurch bleibt immer nur ein kleiner Teil des Programmes sichtbar.

Die beiden am Ende dieses Unterkapitels abgedruckten Programme leisten das Gewünschte. Das erste Programm codiert ein beliebiges Maschinenprogramm. Dieses muß als ersten Befehl 'JSR $C000' enthalten, um die Decodierroutine zu aktivieren. Die ersten sechs Bytes bleiben uncodiert, damit das Programm überhaupt starten kann.

Die verwendete Codierung sieht so aus, daß zu dem aktuellen Byte das LOW-Byte seiner Adresse hinzuaddiert und das Ergebnis mit 'EOR #$55' verknüpft wird. Sie können selbstverständlich eine wesentlich komplexere Codierung verwenden.

Das zweite Programm ist ein sogenannter Einzelschritt-Decodierer. Es verwendet den IRQ-Timer A, um immer dann eine Unterbrechung zu erzeugen, wenn gerade ein Befehl abgearbeitet worden ist. Dann kann der letzte Befehl codiert und der nächste Befehl decodiert werden.

Bei dem zu codierenden Programm gibt es einiges zu beachten. Wie schon gesagt muß es mit 'JSR $C000' beginnen. Es darf kein selbstmodifizierender Code verwendet werden, außer wenn man dabei die Codierung beachtet. Datentabellen können nicht mitcodiert werden. Man sollte keine allzu langen Schleifen verwenden, da die Ablaufgeschwindigkeit drastisch gesenkt wird. Die Tastaturabfrage bleibt während des Programms abgeschaltet. Betriebssystemroutinen können aufgerufen werden, da das ROM

weder codiert noch decodiert wird. Allerdings werden dabei Daten, die sich im RAM-Bereich unter dem ROM befinden, teilweise zerstört.

Der Befehl 'SEI' führt zum Abbruch des Codierens/Decodierens, wodurch man zeitkritische Teile uncodiert in den Programmtext einfügen kann. Dabei müssen drei Bytes nach dem 'SEI' mitcodiert werden. Mit CLI nimmt der Einzelschritt-Decodierer seine Arbeit wieder auf. Wenn Sie die Einzelschritt-Ebene verlassen wollen, so müssen Sie Ihr Programm mit folgenden drei Befehlen abbrechen:

```
SEI
JSR $FF84
JSR $FF8A
```

Vergessen Sie bitte nicht, daß die Bytes des Befehls 'JSR $FF84' mitcodiert werden sollten. Wenn die Codierroutine Sie nach der "Endadresse+1" fragt, geben Sie die Adresse des Befehls 'FF8A' an. Noch sicherer wird das Programm, wenn man 'illegale Opcodes' (Kapitel 3.4) verwendet, die sich problemlos verarbeiten lassen.

Zur Bedienung ist folgendes zu sagen: geben Sie zuerst den BASIC-Loader der Codierroutine ein und starten Sie ihn mit 'RUN'. Laden Sie nun Ihr zu codierendes (Maschinen-) Programm. Die Codierroutine wird mit 'SYS 49152' gestartet. Nachdem Sie die Fragen nach der Start- und Endadresse beantwortet haben, wird Ihr Programm verschlüsselt. Bevor Sie es laufenlassen können, muß sich zuerst der Einzelschritt-Decodierer im Speicher befinden. Wenn Sie den entsprechenden BASIC-Loader benutzt haben und Ihr Programm kurz dahinter steht, zum Beispiel ab $C100, dann können Sie gleich beides als ein Programm abspeichern.

Codierroutine: BASIC-Loader:

```
100 FORI=1TO156STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATAA9,74,A0,C0,20,1E,AB,20,3E,C0,18,69,06,85,FB, 139
301 DATA8A,69,00,85,FC,A9,85,A0,C0,20,1E,AB,20,3E,C0, 9
302 DATA85,FD,86,FE,A0,00,B1,FB,20,96,C0,91,FB,E6,FB, 53
303 DATAD0,02,E6,FC,A5,FB,C5,FD,D0,ED,A5,FC,C5,FE,D0, 7
304 DATAE7,60,20,42,C0,AA,20,52,C0,0A,0A,0A,0A,8D,50, 74
305 DATAC0,20,52,C0,09,00,60,20,CF,FF,38,E9,30,90,0F, 57
306 DATAC9,0A,90,0A,E9,07,C9,0A,90,05,C9,10,B0,01,60, 175
307 DATAA2,F8,9A,A9,3F,20,D2,FF,4C,74,A4,93,53,54,41, 236
308 DATA52,54,41,44,52,45,53,53,45,3A,20,24,00,0D,45, 125
309 DATA4E,44,41,44,52,45,53,53,45,2B,31,3A,20,24,00, 115
310 DATA18,65,FB,49,55,60,00,00,00,00,00,00,00,00,00, 118
```

Einzelschritt-Decodierer: BASIC-Loader

```
100 FORI=1TO132STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,A9,37,8D,14,03,A9,C0,8D,15,03,68,AA,18,69, 157
301 DATA01,85,AC,68,A8,69,00,85,AD,98,48,8A,48,A9,16, 78
302 DATA8D,04,DC,A9,00,8D,05,DC,A9,19,8D,0E,DC,AD,0D, 119
303 DATADC,A2,02,CA,D0,FD,24,FF,58,60,A0,00,B1,AC,20, 15
304 DATA6C,C0,91,AC,C8,C0,03,D0,F4,BA,BD,05,01,85,AC, 102
305 DATABD,06,01,85,AD,A0,00,B1,AC,20,78,C0,91,AC,C8, 80
306 DATAC0,03,D0,F4,A9,11,8D,0E,DC,AD,0D,DC,68,A8,68, 198
307 DATAAA,68,40,18,65,AC,8C,74,C0,18,69,FF,49,55,60, 185
308 DATA49,55,8C,7F,C0,38,E9,FF,38,E5,AC,60,00,00,00, 178
```

Codierroutine: Assemblerlisting

benutzte Register:
$FB/$FC: Zeiger auf zu codierendes Programm
$FD/$FE: Zeiger auf Endadresse + 1

```
C000 LDA #$74        Text 1: LOW-Byte von $C074
C002 LDY #$C0        Text 1: HIGH-Byte von $C074
C004 JSR $AB1E       Text ausgeben
C007 JSR $C03E       Eingabe einer Hexadezimalzahl nach A/X
C00A CLC             Carry für Addition löschen
C00B ADC #$06        Sechs addieren
C00D STA $FB         in Zeiger auf Programm LOW speichern
C00F TXA             HIGH-Byte der Eingabe nach Akku
C010 ADC #$00        Übertrag addieren
C012 STA $FC         in Zeiger auf Programm HIGH speichern
C014 LDA #$85        Text 2: LOW-Byte von $C085
C016 LDY #$C0        Text 2: HIGH-Byte von $C085
C018 JSR $AB1E       Text ausgeben
C01B JSR $C03E       Eingabe einer Hexadezimalzahl nach A/X
C01E STA $FD         LOW-Byte speichern
C020 STX $FE         HIGH-Byte speichern
C022 LDY #$00        Index auf Null setzen
C024 LDA ($FB),Y     Programm-Byte holen
C026 JSR $C096       Byte codieren
C029 STA ($FB),Y     und wieder wegspeichern
C02B INC $FB         Zeiger LOW-Byte erhöhen
C02D BNE $C031       verzweige, wenn kein Übertrag
C02F INC $FC         Zeiger HIGH-Byte erhöhen
C031 LDA $FB         Zeiger LOW-Byte holen
C033 CMP $FD         mit Endadresse LOW vergleichen
C035 BNE $C024       verzweige, wenn ungleich
C037 LDA $FC         Zeiger HIGH-Byte holen
C039 CMP $FE         mit Endadresse HIGH vergleichen
C03B BNE $C024       verzweige, wenn ungleich
C03D RTS             Rücksprung
```

Eingabe einer vierstelligen Hexadezimalzahl:

```
C03E JSR $C042      zweistellige Hexzahl holen
C041 TAX            und als HIGH-Byte nach X schieben
C042 JSR $C052      eine Hexziffer holen
C045 ASL            durch viermaliges
C046 ASL            Links-SHIFTen ins
C047 ASL            obere Halbbyte
C048 ASL            bringen
C049 STA $C050      und für OR-Verknüpfung speichern
C04C JSR $C052      eine Hexziffer holen
C04F ORA #$00       mit oberem Halbbyte verknüpfen
C051 RTS            Rücksprung
```

Eine Hexziffer holen und umwandeln:

```
C052 JSR $FFCF      BASIN: eingegebenes Zeichen holen
C055 SEC            Carry für Subtraktion setzen
C056 SBC #$30       $30 (=48) abziehen
C058 BCC $C069      verzweige, wenn zu klein
C05A CMP #$0A       mit $0A (=10) vergleichen
C05C BCC $C068      verzweige, wenn kleiner
C05E SBC #$07       Sieben subtrahieren
C060 CMP #$0A       mit $0A (=10) vergleichen
C062 BCC $C069      verzweige, wenn zu klein
C064 CMP #$10       mit $10 (=16) vergleichen
C066 BCS $C069      verzweige, wenn zu groß
C068 RTS            Rücksprung
C069 LDX #$F8       X-Register mit $F8 (=248) laden
C06B TXS            Stapelzeiger initialisieren
C06C LDA #$3F       ASCII "?"
C06E JSR $FFD2      Fragezeichen ausgeben
C071 JMP $A474      zurück zum BASIC-Interpreter

Text 1: C074: "STARTADRESSE: $"
Text 2: C085: "ENDADRESSE+1: $"
```

```
C074 93 53 54 41 52 54 41 44 .STARTAD
C07C 52 45 53 53 45 3A 20 24 RESSE: $
C084 00 0D 45 4E 44 41 44 52 ..ENDADR
C08C 45 53 53 45 2B 31 3A 20 ESSE+1:
C094 24 00 18 65 FB 49 55 60 $.......
```

Unterroutine zum Codieren:

```
C096 CLC          Carry für Addition löschen
C097 ADC $FB      Programmzeiger LOW-Byte addieren
C099 EOR #55      mit EXOR #$55 (=85) verknüpfen
C09B RTS          Rücksprung
```

Einzelschritt-Decodierer: Assemblerlisting

benutzte Register:
$AC/$AD: Zwischenspeicher für den Programmzeiger

```
C000 SEI          Interrupts sperren
C001 LDA #$37     LOW-Byte von $C037
C003 STA $0314    in IRQ-Vektor LOW speichern
C006 LDA #$C0     HIGH-Byte von $C037
C008 STA $0315    in IRQ-Vektor HIGH speichern
C00B PLA          Rücksprungadresse LOW vom Stapel holen
C00C TAX          zwischenspeichern
C00D CLC          Carry für Addition löschen
C00E ADC #$01     Eins addieren
C010 STA $AC      und in Programmzeiger LOW speichern
C012 PLA          Rücksprungadresse HIGH vom Stapel holen
C013 TAY          zwischenspeichern
C014 ADC #$00     Übertrag addieren
C016 STA $AD      und in Programmzeiger speichern
C018 TYA          Rücksprungadresse HIGH-Byte
C019 PHA          wieder auf Stapel legen
C01A TXA          Rücksprungadresse LOW-Byte
C01B PHA          wieder auf Stapel legen
C01C LDA #$16     LOW-Byte von $0016 (=22)
C01E STA $DC04    in IRQ-Timer LOW speichern
```

```
C021 LDA #$00       HIGH-Byte von $0016 (=22)
C023 STA $DC05      in IRQ-Timer HIGH speichern
C026 LDA #$19       Akku mit $19 (=25) laden
C028 STA $DC0E      und Timer mit neuem Zählwert starten
C02B LDA $DC0D      Interrupt-Kontrollregister löschen
C02E LDX #$02       Zähler für Zeitschleife setzen
C030 DEX            Zähler verringern
C031 BNE $C030      verzweige, wenn ungleich Null
C033 BIT $FF        drei Taktzyklen warten
C035 CLI            Interrupts erlauben
C036 RTS            Rücksprung
```

Interruptroutine des Einzelschritt-Decodierers:

```
C037 LDY #$00       Index auf Null setzen
C039 LDA ($AC),Y    Programm-Byte holen
C03B JSR $C06C      Byte codieren
C03E STA ($AC),Y    und wieder speichern
C040 INY            Programmzeiger erhöhen
C041 CPY #$03       schon drei Bytes codiert?
C043 BNE $C039      verzweige, wenn nein
C045 TSX            Stapelzeiger ins X-Register
C046 LDA $0105,X    Programmzeiger LOW-Byte holen
C049 STA $AC        und zwischenspeichern
C04B LDA $0106,X    Programmzeiger HIGH-Byte holen
C04E STA $AD        und zwischenspeichern
C050 LDY #$00       Index auf Null setzen
C052 LDA ($AC),Y    Programm-Byte holen
C054 JSR $C078      Byte decodieren
C057 STA ($AC),Y    und wieder speichern
C059 INY            Programmzeiger erhöhen
C05A CPY #$03       schon drei Bytes codiert?
C05C BNE $C052      verzweige, wenn nein
C05E LDA #$11       Akku mit $11 (=17) laden
C060 STA $DC0E      und Timer wieder starten
C063 LDA $DC0D      Interrupt-Kontrollregister löschen
C066 PLA            Y-Register vom Stapel holen
C067 TAY            und übernehmen
C068 PLA            X-Register vom Stapel holen
```

```
C069 TAX          und übernehmen
C06A PLA          Akku vom Stapel holen
C06B RTI          Rückkehr vom Interrupt
```

Unterroutine zum Codieren:

```
C06C CLC          Carry für Addition löschen
C06D ADC $AC      Programmzeiger LOW-Byte addieren
C06F STY $C074    Y-Register für Addition speichern
C072 CLC          Carry für Addition löschen
C073 ADC #$FF     Y-Register addieren
C075 EOR #$55     mit EXOR #$55 (=85) verknüpfen
C077 RTS          Rücksprung
```

Unterroutine zum Decodieren:

```
C078 EOR #$55     mit EXOR #$55 (=85) verknüpfen
C07A STY $C07F    Y-Register für Subtraktion speichern
C07D SEC          Carry für Subtraktion setzen
C07E SBC #$FF     Y-Register subtrahieren
C080 SEC          Carry für Subtraktion setzen
C081 SBC $AC      Programmzeiger LOW-Byte subtrahieren
C083 RTS          Rücksprung
```

3.3.5 Codierung über ASCII-Code

Eine weitere sehr raffinierte Methode zum Codieren von Daten wollen wir Ihnen jetzt vorstellen. Sie zeichnet sich nicht nur durch ihren komplexen Code, sondern auch durch das verblüffende Entcodierungssystem aus.

Der Vorteil bei diesem System ist, daß der Knacker die Entcodierungsroutine kaum als solche erkennt und nur schwer herausfinden kann, wo die entcodierten Daten abgelegt werden. Doch kommen wir jetzt zur Erlärung des Systems.

Die Grundüberlegung beruht auf der Tatsache, daß ASCII-Zeichen, die mit der BSOUT-Routine ($FFD2) auf den Bildschirm geschrieben werden, zuvor in den Bildschirmcode gewandelt werden. Wäre es nicht möglich, entsprechend codierte Programme mit Hilfe der BSOUT-Routine in die Bildschirm zu schicken, und dort zu starten? Durch Setzen des Cursers könnte man die Adresse, zu der die Daten geschoben werden, genau angeben. Kaum ein Knacker würde vermuten, daß durch eine harmlose Ausgabe von Zeichen auf dem Bildschirm ein Programm geschrieben wird. Das hört sich alles fantastisch an und ist mit ein paar Tricks in die Praxis umzusetzen. Das ganze hat leider noch einen kleinen Haken, denn es ist nicht möglich, jedes Byte in einen dazugehörigen ASCII-Code zu wandeln, weil es außer den normalen Zeichen auch Steuer-Codes gibt, die nicht in den Bildschirm-Speicher geschrieben, sondern ausgeführt werden. Außerdem stellt sich das Problem, daß Bildschirm-Codes, die größer als $7F sind, durch dieselben ASCII-Codes wie die Bildschirm-Codes, die kleiner als $80 sind, erzeugt werden, nur mit dem Unterschied, daß Bildschirm-Codes größer $7F mit gesetztem REVERSE-Flag erreicht werden. Zur Verdeutlichung zeigen wir Ihnen dies noch an einem Beispiel.

ASCII-Code	REVERSE	Bildschirm-Code
$41	aus	$01
$41	ein	$81
$93	xxx	Steuer-Code (Bildschirm löschen)

Um diese Probleme zu lösen, haben wir uns den Zusammenhang zwischen ASCII-Code und Bildschirm-Code näher angesehen und daraufhin unseren eigenen ASCII-Code entwickelt.

Bei der Umwandlung von ASCII-Code in Bildschirm-Code ändert sich nur das HIGH-Nibbel, wie man aus folgender Tabelle entnehmen kann:

ASCII-Code	Bildschirm-Code
$40	$00
$50	$10
$20	$20
$30	$30
$C0	$40
$D0	$50
$A0	$60
$B0	$70

Die Bildschirm-Codes größer als $7F werden mit unserer Routine anders gewandelt, als es das Betriebssystem machen würde:

ASCII-Code	Bildschirm-Code
$E0	$80
$F0	$90
$60	$A0
$70	$B0
$80	$C0
$90	$D0
$00	$E0
$10	$F0

Eine entsprechende Ausgabe-Routine wandelt unsere speziellen ASCII-Codes in die normalen Codes um und bringt diese mit gesetztem REVERSE-Flag auf den Bildschirm. Mit diesem Trick kann man alle möglichen Bildschirm-Codes in ASCII-Codes und umgekehrt wandeln. Durch Änderung des HIGH-Bytes der Bildschirm-Anfangsadresse und durch Positionierung des Cursers kann man Daten in jeden beliebigen Speicherbereich schreiben. Um die zu codierenden Daten noch stärker zu "verstümmeln" haben wir jedes Byte zusätzlich mit $2F addiert.

Unser nun folgendes Codierungsprogramm wandelt das sich im Akku befindliche Zeichen in den entsprechenden Code um. X- und Y-Register werden gerettet.

```
1000 STX $FE      X-Register retten
1002 STA $FD      Akku zwischenspeichern
1004 AND #$F0     HIGH-Nibbel isolieren
1006 LDX #$0F     Zeiger auf Tabelle setzen
1008 CMP $101D,X  Zeichen mit Tabelle vergleichen
100B BEQ $1010    verzweige, wenn Zeichen gefunden
100D DEX          Zeiger verringern
100E BPL $1008    unbedingter Sprung
1010 LDA $FD      Akku wieder holen
1012 AND #$0F     HIGH-Nibbel löschen
1014 ORA $102D,X  und mit neuem HIGH-Nibbel verknüpfen
1017 CLC          Carry für Addition löschen
1018 ADC #$2F     Byte mit $2F addieren
101A LDX $FE      X-Register wieder holen
101C RTS          Rücksprung
101D BRK

101E 10 20 30 40 50 60 70 80  Daten, mit denen das
1026 90 A0 B0 C0 D0 E0 F0 40  Byte verglichen wird

102E 50 20 30 C0 D0 A0 B0 E0  entsprechende HIGH-Nibbel
1036 F0 60 70 80 90 00 10 00  die eingesetzt werden
```

Die Umrechnung der Startadresse in unserer Entcodierungs-Routine verfügt zusätzlich über eine Umrechnung der gewünschten Adresse in die dazugehörige Curser-Position. Die Adresse gibt an, wohin das nächste Zeichen durch die BSOUT-Routine geschrieben werden soll. Somit kann man Daten an jeden beliebigen Platz des Speichers schreiben und ist nicht auf den momentan vom sichtbaren Bildschirm beanspruchten Speicherbereich angewiesen. Sie kann auch Daten nach $C000 schreiben, obwohl das Bildschirm-RAM immer noch ab $0400 liegt. Die Adresse, ab der die Daten geschrieben werden sollen, muß im Akku und X-Register stehen, wobei im Akku das LOW- und im X-Register das HIGH-Byte stehen muß. Die Routine liegt ab $112F. Das Byte, das entcodiert und geschrieben werden soll, muß sich im Akku befinden, wenn Sie die Ausgabe-Routine ab $1100 anspringen.

Das folgende Programm erfüllt alle diese Aufgaben:

```
1100 STX $FE      X-Register retten
1102 SEC          Carry für Subtraktion setzen
1103 SBC #$2F     $2F vom Byte abziehen
1105 STA $FD      und Wert zwischenspeichern
1107 AND #$F0     HIGH-Nibbel isolieren
1109 LDX #$07     Zeiger auf Tabelle setzen
110B CMP $1145,X  Wert mit Byte aus Tabelle vergleichen
110E BEQ $1118    verzweige, wenn Wert gefunden
1110 DEX          Zeiger verringern
1111 BPL $110B    verzweige, wenn nicht ganz verglichen
1113 LDA $FD      Zeichen laden
1115 CLC          Carry löschen
1116 BCC $1123    unbedingter Sprung

1118 LDA #$01     REVERSE-Flag
111A STA $C7      setzen
111C LDA $FD      Zeichen laden
111E AND #$0F     LOW-Nibbel isolieren
1120 ORA $114D,X  und mit neuem Nibbel verknüpfen
1123 JSR $FFD2    Zeichen ausgeben
1126 LDA #$00     REVERSE-Flag
1128 STA $C7      wieder löschen
112A LDX $FE      X-Register wieder herstellen
112C RTS          Rücksprung
112D NOP
112E NOP
```

Routine zum Errechnen der Curser-Position:

```
112F STX $0288    HIGH-Byte der Adresse speichern
1132 LDX #$00     X-Register für Division löschen
1134 CMP #$28     LOW-Byte
1136 BCC $113E    durch 40
1138 SEC          teilen
1139 SBC #$28     im X-Register steht das
113B INX          Ergebnis der
```

```
113C BCS $1134   Division
113E STX $D6     Ergebnis als Curser-Zeile nehmen
1140 STA $D3     Rest ist Curser-Spalte
1142 JMP $E56C   Curser positionieren
1145 BRK

1146 10 60 70 80 90 E0 F0 A0  Tabelle unserer eigenen
                              ASCII-Codes

114E B0 20 30 C0 D0 40 50 00  Tabelle zur Umrechnung in
                              normalen ASCII-Code
```

3.4 Die Illegal-Codes

Im nun folgenden Kapitel wollen wir eine spezielle Art der Operations-Codes (kurz Opcodes genannt) besprechen. Es handelt sich hierbei um die illegalen Opcodes, die bei der Programmierung des C64 normalerweise nicht berücksichtigt werden, da sie nicht zu den offiziellen Assemblerbefehlen gehören.

3.4.1 Erklärung der Illegal-Opcodes

Wenn Sie schon öfter in Maschinensprache programmiert haben, wird Ihnen beim Disassemblieren einiger Speicherbereiche bestimmt schon aufgefallen sein, daß dabei nicht nur Assemblerbefehle auf den Bildschirm gebracht werden. Ab und zu mogeln sich auch drei Fragezeichen zwischen die Befehle, was dann folgendermaßen aussehen kann:

```
12F0 LDA $0830
12F3 ???
12F4 INX
```

Dieses 'Fragezeichen-Phänomen' kann beim ersten Betrachten Verwirrung stiften, läßt sich aber relativ einfach erklären: Ein Assemblerbefehl kann zwar aus mehreren Bytes bestehen, so benötigt zum Beispiel LDA #$40 zwei Bytes; der eigentliche Befehl, also LDA, besteht aber immer nur aus einem Ein-Byte-

Codewert. Beim Disassemblieren wird dieser Codewert, hier A9, gelesen und der entsprechende Befehl auf den Bildschirm gebracht.

Mit einem Byte ließen sich also theoretisch 256 verschiedene Befehle definieren. Da die tatsächliche Anzahl der Befehle aber wesentlich kleiner ist, bleiben einige Codewerte unbenutzt, also auch undefiniert. Man spricht hier von undefinierten Operationscodes oder von illegalen Opcodes.

Beim Versuch, diese Opcodes zu disassemblieren, wird kein entsprechender Befehl gefunden und stattdessen nur drei Fragezeichen auf dem Bildschirm ausgegeben.

Daß die illegalen Opcodes in der offiziellen Programmierung der 6510-Maschinensprache keine Verwendung finden, heißt aber nicht, daß sie keine Funktionen erfüllen. Vielmehr ist hier das Gegenteil der Fall, was Tab. 2.4.1 zeigt. Hier ist die erste der drei Gruppen, in die sich die illegalen Opcodes aufteilen lassen, abgebildet. Trifft der Prozessor auf diese Opcodes, stürzt der Rechner ab, oder sie werden einfach überlesen, ohne eine Funktion auszuführen, wie es von dem Opcode $EA (NOP) her bekannt ist.

Illegal Codes

Ab-sturz	NOP 1 Byte	NOP 2 Byte	NOP 3 Byte
02	1A	04	0C
12	3A	14	1C
22	5A	34	3C
32	7A	44	5C
42	DA	54	7C
52	FA	64	DC
62		74	FC
72		80	
92		82	
B2		89	
D2		C2	
F2		D4	
		E2	
		F4	

Abb. 3.4.1: Die erste Gruppe der Illegal-Codes

Die zweite Gruppe verbindet jeweils zwei bekannte Opcodes miteinander. Hierbei können alle Adressierungsarten verwendet werden, die von dem Befehl STA her bekannt sind.

Befehl / Adressierung	ASL: ORA	ROL: AND	LSR: EOR	ROR: ADC	DEC: CMP	INC: SBC	LDA: TAX
(,X)	03	23	43	63	C3	E3	A3
(),Y	13	33	53	73	D3	F3	B3
ABS	0F	2F	4F	6F	CF	EF	AF
ABS,X	1F	3F	5F	7F	DF	FF	/
ABS,Y	1B	3B	5B	7B	DB	FB	BF
ZP	07	27	47	67	C7	E7	A7
ZP,X	17	37	57	77	D7	F7	/
ZP,Y	/	/	/	/	/	/	B7

Abb. 3.4.2: **Die zweite Gruppe der Illegal-Codes**

Die Befehle der dritten Gruppe lassen sich durch bekannte Opcodes nur sehr schwer ersetzen. Sie führen, wie die Befehle der zweiten Gruppe auch, zwei oder mehr Befehle auf einmal aus. Da es sich dabei zumeist um recht komplexe Operationen handelt, ist der Einsatzbereich ziemlich begrenzt. Es sei hier noch erwähnt, daß die illegalen Opcodes nur unbeabsichtigte Nebenprodukte des eigentlichen Befehlssatzes sind und aus diesem Grund auch nicht bei jedem Computer die gleiche Wirkung erzielen.

ILLEGAL-CODES

0B MM	AND MM und bringt Negativflag ins Carry
2B MM	wie 0B MM
4B MM	AND #MM : LSR
6B MM	wenn Dezimal = 0: AND #MM : ROR: ausserdem kommt Bit 0 des Akkus ins Carry und Bit 5 EOR mit Bit 6 ins Overflow
83 MM	Akku AND mit X-Register kommt nach (MM,X)
87 MM	Akku AND mit X-Register kommt nach MM
8B MM	TXA : AND #MM
8F LO HI	Akku AND mit X-Register kommt nach LO HI
93 MM	AND zwischen Akku, X-Register und die Summe aus 1 und MM+1 kommt nach (MM),Y
97 MM	Akku AND X-Register kommt nach MM,X
9B LO HI	Akku AND mit X-Register in Stapelzeiger, dann Stapelzeiger AND #HI+1 nach LO HI,Y
9C LO HI	Y-Register AND mit #HI+1 nach HI LO,X
9E LO HI	X-Register AND mit #NN+1 nach HI LO,Y
9F LO HI	AND zwischen Akku, X-Register, #HI+1 nach HI LO,Y
BB LO HI	HI LO,Y AND mit Stapelzeiger ins X-Register, dann TXS:TXA
CB MM	Akku AND mit X-Register ins X-Register, dann X-Register minus #MM (ohne Carry)
EB MM	SBC #MM

Abb. 3.4.3: Dritte Gruppe der Illegal-Codes

3.4.2 Anwendung der Illegal-Opcodes

Die Verwendung von undefinierten Opcodes in einem Programm bringt für den Programmierer drei Vorteile mit sich. Als erstes wäre zu erwähnen, daß sich bestimmte Befehlsfolgen durch Illegal-Opcodes kürzer, das heißt mit weniger Bytes darstellen lassen. Zu diesem Zweck sind diese Opcodes aber denkbar ungeeignet, da das Programm nur unwesentlich kürzer wird, der Programmieraufwand aber erheblich ansteigt, da die Codes direkt als Hexwerte eingegeben werden müssen.

Der zweite Vorteil liegt darin, daß für einige Befehle bzw. Befehlskombinationen neue Adressierungsarten verwendet wer-

den können, die normalerweise nicht zur Verfügung stehen. Aber auch hier ist der Aufwand so hoch, daß man besser auf die Standardbefehle zurückgreifen sollte.

Der dritte und einzig wesentliche Vorteil besteht darin, daß die Illegal-Opcodes von einen normalen Disassembler nicht gelesen und in Befehlsworte "übersetzt" werden können. Dadurch wird das Lesen der Programme für Unbefugte erheblich erschwert. Verfügt man nicht über die oben abgebildeten Tabellen, wird dieses sogar zu einer unüberwindlichen Hürde. Wollen Sie also wichtige Teile Ihres Programms vor Fremdzugriffen schützen, bieten die Illegal-Opcodes eine große Hilfe.

Es ist zum Beispiel äußerst sinnvoll, die Abfrage des Kopierschutzes auf Diskette durch Illegal-Opcodes zu verstecken. Ein weiteres Beispiel bezieht sich auf die Decodierung von Programmen. Hier könnte man die Decodierroutine für das Hauptprogramm nochmals codieren. Dieses erfordert dann eine weitere, jedoch sehr kurze Decodierroutine, die mit Hilfe der Illegal-Opcodes geschrieben werden kann. Ist der Einsprung des Programms nicht bekannt, so ist es nahezu unmöglich, überhaupt eine Codier- bzw. Decodierroutine zu entdecken. Es ist also auch nicht möglich, das Programm zu decodieren und dann zu verändern.

Die einfachste Methode, ein Programm durch Illegal-Opcodes unleserlich, also auch unverständlich, zu gestalten, besteht darin, zwischen einzelne Befehle einfach Illegal-Opcodes zu setzen, die ein NOP ausführen. Diese Opcodes finden Sie in Abbildung 3.4.1. Wesentlich besser ist es jedoch, das Programm selbst mit Illegal-Opcodes zu schreiben.

Wie es möglich ist, ein ganz simples Programm mit Hilfe der Illegal-Opcodes so zu verschlüsseln, daß es für jeden Unbefugten unlesbar wird, wollen wir an dem nun folgenden Beispiel demonstrieren. Wir haben ein kleines Programm geschrieben, das durch zwei ineinander verschachtelte Schleifen die Rahmenfarbe verändert. Hierbei dient die Speicherzelle $FB (251) als Zähler für die innere Schleife und das X-Register als Zähler für die äußere Schleife.

```
1000 LDX #$00      Zeiger für äußere Schleife
1002 LDA #$00      Wert für innere Schleife
1004 STA $FB       Zähler setzen
1006 DEC $D020     Rahmenfarbe vermindern
1009 DEC $FB       Zeiger vermindern
100B LDA $FB       Zeiger laden
100D CMP #$00      innere Schleife beendet?
100F BNE $1006     wenn nein, dann weiter
1011 INX           Zeiger für äußere Schleife erhöhen
1012 BNE $1006     wenn kleiner 256, dann weiter
1014 BRK           Programmende
```

Bei der ersten Modifikation werden die drei Befehle in $1009, $100B und $100D, die den inneren Schleifenwert vermindern und auf Null überprüfen, durch den Illegal Opcode $C7 ersetzt. Diesen Befehl finden Sie in der zweiten Tabelle (Abb. 3.4.2). Er bewirkt ein DEC $ZEROPAGE-ADRESSE; CMP $ZERO-PAGE-ADRESSE.

```
1000 LDX #$00      Zeiger der äußeren Schleife
1002 LDA #$00      Zeiger der inneren
1004 STA $FB       Schleife setzen
1006 DEC $D020     Farbe vermindern
1009 ???           Illegal $C7 (DEC) und
100A ???           Adresse $FB (+CMP)
100B BNE $1006     zum Schleifenanfang
100D INX           Zeiger erhöhen
100E BNE $1006     zum Schleifenanfang
1010 BRK           Programmende
```

```
1000 A2 00 A9 00 85 FB CE 20
1008 D0 C7 FB D0 F9 E8 D0 F6
1010 00 00 00 00 00 00 00 00
```

Wie Sie sehen, wird schon durch diese eine Programmänderung das gesamte Konzept recht unübersichtlich.

Bei der zweiten Änderung wird das Laden des ersten Zeigers und des Wertes für den zweiten Zeiger durch den Opcode $AF ersetzt. Dadurch wird der Wert aus der Adresse in den Akku geladen, die sich aus den beiden folgenden Bytes ergibt. Danach wird der Inhalt des Akkus in das X-Register geschoben. Da wir die beiden Zeiger auf $00 setzen wollen, müssen wir aus einer Adresse laden, in der immer ein Null-Byte steht. Wir haben dafür die Adresse $E379 gewählt, da diese sich im ROM-Bereich befindet.

```
1000 ???          Illegal-Opcode
1001 ADC $85E3,Y  führt LDA $E379
1004 ???          und TAX aus
1005 DEC $D020    Farbe vermindern
1008 ???          Illegal-Opcode führt
1009 ???          DEC $FB:CMP $FB aus.
100A BNE $1005    zum Schleifenanfang
100C INX          Zeiger erhöhen
100D BNE $1005    zum Schleifenanfang
100F BRK          Programmende
```

```
1000 AF 79 E3 85 FB CE 20 D0
1008 C7 FB D0 F9 E8 D0 F6 00
1010 00 00 00 00 00 00 00 00
```

Wenn Sie sich das disassemblierte Programm anschauen, stellen Sie fest, daß nicht nur der LDA- und der LDX-Befehl, sondern auch der STA-Befehl nicht mehr zu lesen ist. Das ist dadurch zu erklären, daß beim Disassemblieren der Inhalt der Speicherzelle $1000 nicht verarbeitet werden kann und so drei Fragezeichen ausgegeben werden. Danach wird aus $1001 der Hexwert $79 gelesen, der eigentlich die Zieladresse für den Illegal-Opcode darstellt, und als ADC $....,Y verarbeitet. Die nächsten beiden Bytes werden als Adresse für den ADC-Befehl betrachtet.

So kommt es, daß der STA-Befehl nicht mehr disassembliert wird. Während des Programmablaufs wird dieser Befehl natürlich ordnungsgemäß verarbeitet, da der Prozessor im Gegensatz zum Disassembler mit dem Illegal-Opcode etwas anzufangen weiß.

```
1000 ???
1001 ADC $85E3,Y
1004 ???
1005 ???
1006 JSR $C7D0
1009 ???
100A BNE $1005
100C INX
100D BNE $1005
100F BRK
```

```
1000 AF 79 E3 85 FB CF 20 D0
1008 C7 FB D0 F9 E8 D0 F6 00
1010 00 00 00 00 00 00 00 00
```

Bei der dritten Modifikation unseres Programms wurde der Befehl DEC $D020 durch den Illegal-Opcode $CF ersetzt. Dadurch wird wieder ein DEC $Adresse:CMP $Adresse ausgeführt. Der CMP-Befehl soll uns hier nicht stören, da unmittelbar danach durch den Opcode $C7 noch ein DEC ausgeführt wird, der Zustand der Flags zu diesem Zeitpunkt also unberücksichtigt bleibt.

3.4.3 Taktzyklen der Illegal-Codes

Zum Schluß haben wir für Sie die Taktzyklen der Illegal-Opcodes in Abbildung 3.4.4 zusammengestellt.

Hexcode	Taktzyklus	Hexcode	Taktzyklus	Hexcode	Taktzyklus	Hexcode	Taktzyklus
03	8	4B	2	8F	2	E3	8
04	3	4F	6	93	6	E7	5
07	5	53	8	97	4	EB	2
0B	2	54	4	9B	5	EF	6
0C	4	57	6	9C	5	F3	8
0F	6	5A	2	9E	5	F4	4
13	8	5B	7	9F	5	F7	6
14	4	5C	5	A3	6	FA	2
17	6	5F	7	A7	3	FB	7
1A	2	63	8	AF	4	FC	5
1B	7	64	3	B3	5	FF	7
1C	5	67	5	B7	4		
1F	7	6B	2	BB	5		
23	8	6F	6	BF	5		
27	5	73	8	C2	2		
2B	2	74	4	C3	8		
2F	6	77	6	C7	5		
33	8	7A	2	CB	2		
34	4	7B	7	CF	6		
37	6	7C	5	D3	8		
3A	2	7F	7	D4	4		
3B	7	80	2	D7	6		
3C	5	82	2	DA	2		
3F	7	83	6	DB	7		
43	8	87	3	DC	5		
44	3	89	2	DF	7		
47	5	8B	2	E2	2		

Abb. 3.4.4: **Taktzyklen der Illegal-Codes**

3.5 Dongle als Programmschutz

Bestimmt haben Sie in letzter Zeit schon mal etwas über Dongles gehört oder gelesen. Es handelt sich dabei um einen Hardware-Zusatz, der zu der jeweiligen Programmdiskette mitgeliefert wird.

Programme, die ein Dongle als Kopierschutz verwenden, sind zur Zeit noch recht selten zu finden. Aber schon bald könnte sich dieses Verfahren parallel zum bisher gebräuchlichen Kopierschutz auf Diskette etablieren. Der Vorteil besteht darin, daß auch eine zum Original absolut identische Kopie ohne das Dongle nicht lauffähig ist. Zu einem einfachen Kopierschutz existiert zumeist schon ein Programm, mit dem man eine Sicherheitskopie der Diskette anfertigen kann. Ist das nicht der Fall, so erscheint kurze Zeit nach dem Auftauchen des Kopierschutzes ein Kopierprogramm, das diese Aufgabe sogar mit "Garantie"

übernimmt. Dongles jedoch lassen sich nicht einfach mit einem Kopierprogramm vervielfältigen.

Der Aufwand, der für den Nachbau eines Dongles erforderlich ist, überschreitet meistens die Arbeitsbereitschaft und vor allem das Fachwissen der Raubkopierer. So kann das Dongle unter Umständen einen idealen Kopierschutz darstellen.

3.5.1 Wie arbeitet ein Dongle?

Die zur Zeit übliche Form des Dongles ist ein kleiner Stecker, der in den Control-Port des Rechners eingesteckt wird. Neben der normalen Kopierschutzabfrage der Diskette wird während des Programmablaufs das Dongle kontrolliert. Ist das Dongle nicht eingesteckt oder handelt es sich um ein falsches Dongle, so wird das Programm abgebrochen. Die Abfrage kann einmalig, an einer bestimmten Stelle des Programms oder in einem immer wieder aufgerufenen Unterprogramm geschehen.

Eine recht einfache Art, ein Dongle zu bauen und auch abzufragen, besteht darin, verschiedene Leitungen des Control-Ports auf Masse zu legen. Dazu einige Erläuterungen zum Control-Port und zur Joystick-Abfrage. Die nachfolgend aufgeführten Eigenschaften beziehen sich auf beide Control-Ports. Die Adressen und das unten abgedruckte Beispielprogramm beziehen sich auf Control-Port 2.

Normalerweise sind die Pins 0 bis 4 des Control-Ports auf HIGH gesetzt, das heißt, sie führen Strom. Direkt abhängig von diesen Zuständen sind die Bits 0 bis 4 in $DC00 (56320). Diese Bits stehen normalerweise also auch auf HIGH, das heißt, sie enthalten eine 1.

Bei einer Bewegung des Joysticks werden die jeweiligen Leitungen mit der Leitung 7 des Control-Ports verbunden und dadurch auf Masse gelegt. Das dementsprechende Bit in $DC00 (56320) wird dadurch gelöscht. Die Bits 0 bis 3 entsprechen üblicherweise den vier möglichen Richtungen. Das Bit 4 ist für den Feuerknopf zuständig.

Durch ein Dongle können nun verschiedene Leitungen des Control-Ports gleichzeitig auf Masse gelegt und abgefragt werden. Bei einem eingegossenen Dongle sind die Verbindungen der einzelnen Leitungen nur noch durch Ausmessen herauszufinden.

3.5.2 Eine einfache Dongle-Abfrage

Im unserem nun folgenden Beispiel fragen wir ein Dongle ab, das die Leitungen 0 und 1 auf Masse legt, also mit Leitung 7 verbindet. Normalerweise sind diese Leitungen beim Joystick für die Richtungen OBEN und UNTEN zuständig, so daß das Dongle mit einem Joystick nicht imitiert werden kann. Um das Dongle nachzubauen, benötigen Sie nur einen Control-Port-Stecker und drei kleine Kabelstücke. Diese Teile erhalten Sie für einige Groschen in jedem Elektronik-Fachgeschäft.

```
10 A$ = "(CLR)O.K."
20 PRINT"(CLR)DONGLE EINSTECKEN UND TASTE DRUECKEN"
30 POKE 198,0:WAIT 198,1
40 A= PEEK(56320) AND 3
50 IF A <>0 GOTO 70
60 PRINT A$:GOTO 40
70 PRINT "ENDE",A
```

Da die Tastaturabfrage des Rechners ebenfalls über die Leitungen des Control-Ports geschieht, kann die Tastatur hier nicht mehr vollständig abgefragt werden. Stecken Sie das Dongle also erst ein, nachdem Sie das Programm gestartet haben.

In Zeile 20 und 30 werden Sie aufgefordert, das Dongle einzustecken. Das Programm wartet nun, bis Sie eine beliebige Taste gedrückt haben. In Zeile 40 wird das Port-Register A aus $DC00 gelesen und die Bits 0 und 1 isoliert. (2^0+2^1 = 1+2 = 3, durch die AND-Verknüpfung mit diesem Wert werden alle anderen Bits gelöscht.)

Wenn die Bits nicht beide auf LOW stehen, also 0 sind, wird das Programm in 70 beendet. In Zeile 60 wird "O.K." auf den Bildschirm gebracht und wieder zum Schleifenanfang verzweigt.

Dieses kleine Programm sollte Ihnen die Arbeitsweise eines Dongles verdeutlichen. Es ist so natürlich noch nicht als Kopierschutz geeignet, da die Abfrage für jedermann einsehbar als BASIC-Programm vorliegt. Wie Sie das Programm gegen fremde Augen schützen, haben wir allerdings in den vorhergehenden Kapiteln schon ausgiebig beschrieben.

3.5.3 Dongle mit IC

Das unter Kapitel 3.5.2 beschriebene Dongle, in dem verschiedene Leitungen gleichzeitig auf Masse gelegt wurden, kann mit einiger Fachkenntnis und den entsprechenden Meßgeräten relativ einfach ausgemessen und nachgebaut werden. Auch wenn die meisten C64-Besitzer nicht über diese Grundvoraussetzungen verfügen und ihnen der Arbeitsaufwand zu groß ist, offenbart sich hier doch eine Schwäche des Systems.

Abhilfe würde hier ein Dongle schaffen, dessen Leitungen nicht fest, sondern variabel auf Masse gelegt werden können. Um das zu erreichen, haben wir in das Dongle ein IC eingesetzt, das eine ganz einfache Schaltung enthält. Prinzipiell kann man dafür fast jede Schaltung benutzen, die aus einfachen oder miteinander kombinierten Gattern besteht. Wie oben erwähnt, wird die Tastatur aber auch über die Port-Leitungen abgefragt, so daß die einzelnen Leitungen möglichst auf HIGH liegen sollten. Daraus ergibt sich, daß zum Beispiel eine Schaltung mit einem NAND-Gatter, also einem NOT-AND-Gatter, ungeeignet wäre, weil hierbei mindestens eine Leitung immer LOW ist.

Wir haben für das Dongle ein IC mit vier voneinander unabhängigen AND-Gattern gewählt, wovon allerdings nur ein Gatter benutzt wird. Das hier verwendete Gatter hat zwei Eingänge und einen Ausgang, der im "Normalzustand" auf HIGH liegt. Die genaue Schaltung können Sie Abbildung 3.5.1 entnehmen.

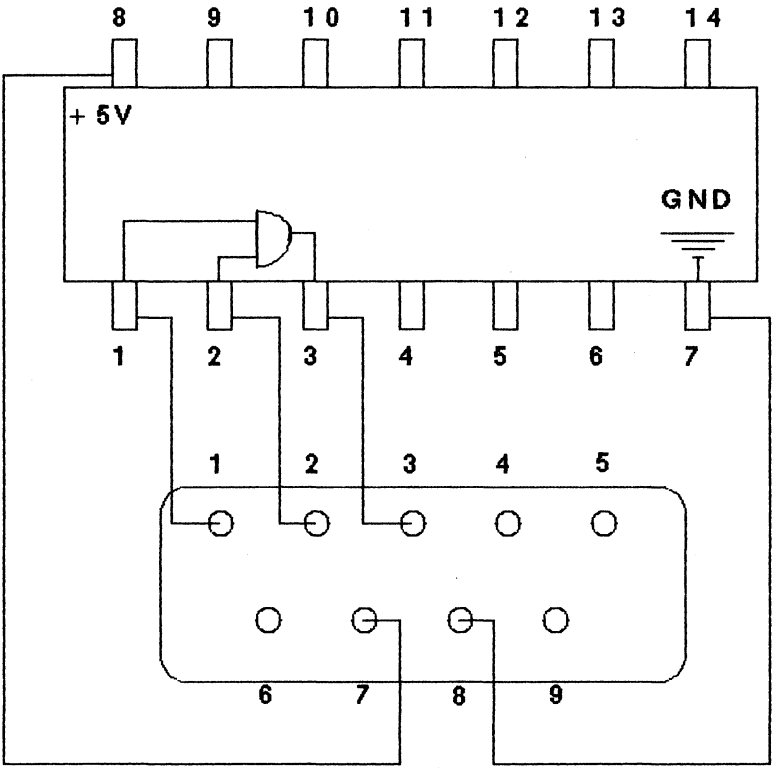


Abb. 3.5.1: Dongle mit AND-Gatter

Hier nun ein Programm, mit dem das Dongle abgefragt werden kann. Dabei wird das Dongle bei jedem Interrupt umgeschaltet und überprüft, so daß jedes andere Programm unabhängig davon laufen kann. Wird ein falsches Dongle erkannt, verzweigt das Programm zum unbedingten RESET. An dieser Stelle können Sie aber auch zu einem eigenen Programm verzweigen, das zu Beispiel den gesamten Speicher löscht und dann einen Absturz verursacht. Hierzu können weitere Informationen zum Beispiel aus Kapitel 3.4 über die Illegal-Opcodes entnommen werden.

Interrupt-Vektor auf Abfrageroutine stellen:

```
C000 SEI        Interrupt verhindern
C001 LDA #$0D   LOW-Byte des neuen Interrupt-Vektors
C003 STA $0314  abspeichern
C006 LDA #$C0   HIGH-Byte des neuen Interrupt-Vektors
C008 STA $0315  abspeichern
C00B CLI        Interrupt wieder freigeben
C00C RTS        Rücksprung
```

Abfrageprogramm für Dongle:

```
C00D PHA        Akku-Inhalt auf Stapel retten
C00E LDA $DC00  Wert für Control-Port 2 holen
C011 PHA        und auf Stapel retten
C012 AND #$FE   Bit 0 löschen
C014 STA $DC00  Wert zurückschreiben
C017 LDA $DC00  neuen Wert laden
C01A AND #$05   Bit 0 und Bit 2 isolieren
C01C BNE $C037  wenn Bit 0 und 2 gesetzt, dann RESET
C01E PLA        Originalwert für Control-Port wiederholen
C01F STA $DC00  und zurückschreiben
C022 LDA $DC00  neuen Wert holen
C025 ORA #$01   Bit 0 setzen
C027 STA $DC00  Wert zurückschreiben und
C02A LDA $DC00  neuen Wert holen
C02D AND #$05   Bit 0 und Bit 2 isolieren
C02F EOR #$05   Bits invertieren
C031 BNE $C037  wenn Bit 0 und 2 nicht gesetzt, RESET
C033 PLA        Akkuinhalt vom Stapel holen
C034 JMP $EA31  zur Betriebssystem-Interruptroutine
```

RESET ausführen, wenn falsches Dongle erkannt wird:

```
C037 INC $D020  Rahmenfarbe erhöhen
C03A SEI        Interrupt verhindern
C03B JMP $FCEF  RESET ohne CBM80- bzw. Modulabfrage
```

Hier nun das Programm als BASIC-Loader:

```
5 S=0:A=0:X=0
10 READ A:S=S+A:IF A=-1 THEN GOTO 25
12 POKE49152+X, A:X=X+1
15 GOTO 10
25 IF S <> 6626 THEN PRINT "FEHLER IN DATA S", S:STOP
30 SYS 49152
100 DATA 120, 169, 13, 141, 20, 3, 169, 192, 141, 21, 3, 88, 96, 72, 173
, 0, 220, 72, 41, 254
110 DATA 141, 0, 220, 173, 0, 220, 41, 5, 208, 25, 104, 141, 0, 220, 173
, 0, 220, 9, 1, 141, 0
120 DATA 220, 173, 0, 220, 41, 5, 73, 5, 208, 4, 104, 76, 49, 234, 238,
32, 208, 120, 76, 239
130 DATA 252, 0, -1
```

Natürlich ist die hier beschriebene Anwendung des Dongles nur ein Beispiel unter vielen. Weitere Abfragen gleicher oder ähnlicher Art können zum Beispiel auch über den User-Port geschehen. Alle möglichen Varianten zu beschreiben, würde allerdings den Rahmen dieses Buches überschreiten, so daß hier Ihre Fantasie angesprochen wird, um weitere Methoden selbst auszuklügeln.

3.6 Password-Abfrage

Eine Password-Abfrage ist eigentlich dafür gedacht, daß ein Unbefugter nicht ohne weiteres bestimmte Programme oder Dateien benutzen kann. Das Password kann entweder direkt am Anfang eines Programms abgefragt werden oder an ganz bestimmten Stellen eines Programms, um diese zu sichern.

Man kann Password-Abfragen in BASIC oder in Maschinensprache schreiben. Die erste Möglichkeit eignet sich nur, wenn

man diese ausreichend schützt, damit man sich das Password nicht einfach über den LIST-Befehl aneignen kann. Wenn man nun eine Password-Abfrage in BASIC schreibt, sollte man diese nachher mit einem LIST-RUN/STOP-RESTORE- und einem Änderungsschutz versehen.

Eine Password-Abfrage kann folgendermaßen aussehen:

```
10 POKE 808,225:PRINT CHR$(147):A=0
20 POKE 19,1
30 PRINT CHR$(19):INPUT "PASSWORD :";A$
40 POKE 19,0:PRINT CHR$(13)
50 IF A$="" THEN 20
60 IF A$="HARALD" THEN PRINT CHR$(17);"PASSWORD AKZEPTIERT!":END
70 A=A+1:IF A=3 THEN SYS 64738
80 GOTO 20
```

Das Programm holt sich per INPUT den Text in den Speicher und vergleicht diesen mit dem vorgegebenen Password. Die POKEs vor und hinter dem INPUT unterdrücken das Fragezeichen, das normalerweise erscheinen würde.

Außerdem ist noch ein Zähler eingebaut, der nach dem dritten falschem Wort einen Systemreset durchführt. Falls einfach nur RETURN gedrückt wird, verzweigt das Programm sofort zum INPUT, ohne vorher den Zähler zu erhöhen.

Falls HARALD eingegeben wird, gibt das Programm die Meldung PASSWORD AKZEPTIERT aus und schließt das Programm mit END ab. Beim Einbau in ein eigenes BASIC-Programm kann an diese Stelle ein GOTO eingesetzt werden, um zu einer eigenen Routine zu verzweigen.

Da aber eine Password-Abfrage in BASIC trotzdem zu unsicher ist, weil man sich den BASIC-Code einfach im Maschinensprachemonitor ansehen könnte, obwohl das BASIC 'listgeschützt' ist, sollte man daher das Password in Maschinensprache abfragen.

Wir haben Ihnen eine Password-Abfrage in Maschinensprache geschrieben. Das Programm liegt ab der Adresse $C000 und wird einfach mit SYS 49152 gestartet.

```
C000 JSR $E544      Bildschirm löschen
C003 LDX #$00       Zähler auf Null
C005 LDA $C063,X    PASSWORD:
C008 JSR $FFD2      ausgeben
C00B INX            Zähler erhöhen
C00C CPX #$09       wenn alle Buchstaben geholt,
C00E BNE $C005      dann weiter
C010 LDX #$00       Zähler auf Null
C012 JSR $FFCF      zur INPUT-Routine verzweigen
C015 STA $C080,X    geholtes Byte speichern
C018 INX            Zähler erhöhen
C01+9 CMP #$0D       auf RETURN warten
C01B BNE $C012      falls gedrückt, dann weiter
C01D DEX            RETURN-Code aus Text löschen
C01E STX $02        Zähler speichern
C020 LDX #$00       Zähler auf Null
C022 LDA $C06C,X    Password-Text
C025 EOR #$24       decodieren
C027 STA $C06C,X    und Speichern
C02A INX            Zähler erhöhen
C02B CPX #$06       wenn alles decodiert,
C02D BNE $C022      dann weiter
C02F LDX #$00       Zähler auf Null
C031 LDA $C080,X    Eingegebenes Password
C034 CMP $C06C,X    mit vorhandenem vergleichen
C037 BEQ $C03C      wenn richtig, dann normal weiter
C039 JMP $C051      ansonsten Password decodieren und RESET
C03C INX            Zähler erhöhen
C03D CPX $02        Länge vergleichen
C03F BNE $C031      wenn alle getestet, dann weiter
C041 LDX #$00       Zähler auf Null
C043 LDA $C06C,X    Password
C046 EOR #$24       wieder decodieren
C048 STA $C06C,X    und speichern
C04B INX            Zähler erhöhen
```

```
C04C CPX #$06       wenn alles decodiert,
C04E BNE $C043      dann weiter
C050 RTS            Rücksprung
C051 LDX #$00       Zähler auf Null
C053 LDA $C06C,X    Password
C056 EOR #$24       decodieren
C058 STA $C06C,X    und speichern
C05B INX            Zähler erhöhen
C05C CPX #$06       wenn alles decodiert
C05E BNE $C053      dann weiter
C060 JMP $FCE2      Systemreset

C063 50 41 53 53 57 4F 52 54 PASSWORD
C06B 3A 00 00 00 00 00 00 00 :.......
```

Nachfolgend der BASIC-Loader:

```
100 FORI=1TO120STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A:NEXT:READA: IFC=ATHENC=0:NEXT
:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA20,44,E5,A2,00,BD,63,C0,20,D2,FF,E8,E0,09,D0, 93
301 DATAF5,A2,00,20,CF,FF,9D,80,C0,E8,C9,0D,D0,F5,CA, 175
302 DATA86,02,A2,00,BD,6C,C0,49,24,9D,6C,C0,E8,E0,06, 23
303 DATAD0,F3,A2,00,BD,80,C0,DD,6C,C0,F0,03,4C,51,C0, 187
304 DATAE8,E4,02,D0,F0,A2,00,BD,6C,C0,49,24,9D,6C,C0, 79
305 DATAE8,E0,06,D0,F3,60,A2,00,BD,6C,C0,49,24,9D,6C, 242
306 DATAC0,E8,E0,06,D0,F3,4C,E2,FC,50,41,53,53,57,4F, 88
307 DATA52,54,3A,6C,65,76,65,68,60,00,00,68,60,00,00, 28
308 DATA00,00,00,00,00,00,00,00,44,4B,44,4B,4A,44,4B, 247
```

Diese Maschinenroutine kann jederzeit aus einem BASIC-Programm heraus angesprungen werden, um das Password abzufragen. Es ist nur eine Eingabe möglich. Sobald etwas falsch eingegeben wird, wird sofort ein Systemreset durchgeführt.

Nun ein wenig zur Funktionsweise des Programms: Das Maschinenprogramm macht eigentlich nichts anderes, als das vorherige in BASIC. Es gibt den Text 'PASSWORD :' auf dem Bildschirm aus und verzweigt anschließend zur INPUT-Routine des Betriebssystems.

Die geholten Zeichen werden in einem Speicherbereich hinter dem Maschinenprogramm abgelegt. Die Länge der Zeichenkette wird in der Speicherzelle $02 zwischengespeichert. Sie ist für die spätere Abfrage des Passwords notwendig.

Gleich darauf wird das vorgegebene Password, in unserem Fall 'HARALD', decodiert, damit ein Vergleich der beiden Passwords möglich wird. Das vorgegebene Password befindet sich ab der Adresse $C06C. Dort kann man sein eigenes Password einsetzen. Es darf nicht länger als 16 Zeichen sein, weil es sonst von einem anderen Teil des Programms überschrieben wird. Beim Ändern des Passwords müssen noch zwei Dinge berücksichtigt werden. Nach dem Ändern des Passwords muß dieses wieder mit EOR #$24 codiert werden, weil das Programm das Password codieren würde, statt es zu decodieren. Das würde unweigerlich zu einer falschen Abfrage führen.

Falls eine Änderung des Passwords notwendig ist, verfahren Sie wie folgt: Tragen Sie das neue Password ab der Adresse $C06C ein, indem Sie die ASCII-Zeichen einfach mit dem Maschinensprachemonitor eintragen oder aus dem BASIC einpoken. Merken Sie sich die Länge des Passwords. Passen Sie dann die Vergleichsbefehle in den Adressen $C02B, $C04C und $C05C an die Länge des neuen Passwords an, und springen Sie in die Adresse $C051 des Password-Programms ein. Dort wird das neue Password codiert. Nachdem dieses geschehen ist, wird ein Systemreset durchgeführt.

Damit haben Sie das Password des Programms geändert und können es somit wieder auf Diskette oder Kassette von $C000 bis $C074 mit dem Maschinensprachemonitor abspeichern.

Es gibt noch eine weitere Möglichkeit, eine Art Password abzufragen, und zwar durch Drücken mehrerer Tasten gleichzeitig.

Das Programm springt nur aus der Schleife, falls die Tasten 'M', '8' und 'D' gleichzeitig gedrückt sind. So eine Abfrage muß natürlich in Maschinensprache erfolgen, da die Tasten direkt vom Port geholt werden.

Die CIAs verfügen über zwei 8-Bit-Datenregister (PRA und PRB), bei denen jede der Leitungen sowohl als Eingang als auch als Ausgang gewählt werden kann. Um dieses festzulegen, verfügen die CIAs über jeweils zwei 8-Bit-Datenrichtungsregister (DDRA und DDRB). Wenn ein Bit im DDR gesetzt ist, also gleich 1 ist, arbeitet das korrespondierende Bit des PRs als Ausgang. Ist das Bit im DDR gelöscht, also gleich Null, so ist das entsprechende Bit des PRs als Eingang definiert. Durch das Auslesen der DDRs erfährt man die Eingangsspannungsbelegungen der PRs.

Die Portleitungen PA0 bis PA7 und PB0 bis PB7 des CIAs 1 fragen unter anderem die Tastatur ab. Sie bilden eine 8mal8-Matrix, die es ermöglicht, 64 Tasten abzufragen.

Falls Sie die Tasten Ihres C64 einmal gezählt haben, werden Sie festgestellt haben, daß dieser 66 Tasten hat. Das ist dadurch zu erklären, daß die RESTORE-Taste über eine eigene Leitung verfügt, die bei einem Tastendruck die RESTORE-Leitung auf Masse legt und dadurch einen NMI, also einen nicht maskierbaren Interrupt, auslöst. Die zweite fehlende Taste ist SHIFT-LOCK. Sie ist parallel zur linken SHIFT-Taste geschaltet.

Eine Erklärung vorweg: Wenn ein Daten-Port im Datenrichtungsregister auf Eingang geschaltet und dieser nicht belegt ist, sind diese Bits im Datenregister auf HIGH geschaltet, also gesetzt. Zum besseren Verständnis arbeiten Sie bitte mit der Tabelle am Ende des Kapitels.

Beim CIA 1 sind die Leitungen PA0 bis PA7 als Ausgang geschaltet, die Leitungen PB0 bis PB7 als Eingang. Fragt die Interrupt-Routine des Betriebssystems die Tastatur ab, so legt sie die Anschlüsse des Ports A kurzzeitig LOW, die Bits der Adresse $DC00 (56320) werden gelöscht.

Wird eine Taste gedrückt, so wird das LOW-Signal des Ports A über die Leiterbahn auf das entsprechende Bit des Ports B übertragen. Die anderen Bits des Ports B sind in diesem Fall nicht angeschlossen und deshalb, wie bereits erklärt, gesetzt. Erkennt das Betriebssystem einen vollständig gesetzten Port B, so wurde keine Taste gedrückt, und es wird zum Ende der Tastenabfrage gesprungen.

Wird beispielsweise H gedrückt, so wird Bit 5 des Port B gelöscht. Daran kann der Rechner erkennen, daß eine der Tasten 'F3', 'S', 'F', 'H', 'K', ':', '=' oder die Commodore-Taste gedrückt sein muß. Um festzustellen, welche dieser Tasten gedrückt ist, setzt der Rechner alle Bits des Ports A bis auf das erste auf HIGH und schiebt die Bits des Ports B nacheinander in das Carry-Flag, um zu überprüfen, ob sie gelöscht sind. Sobald er ein gelöschtes Bit festgestellt hat, holt er sich den entsprechenden Tastencode aus einer der Tabellen ab $EB81. Aus welcher Tabelle der Tastencode geholt wird, hängt davon ab, ob die SHIFT- oder die COMMODORE-Taste gleichzeitig betätigt wurden. Waren alle Bits des Ports B gesetzt, wird das nächste Bit des Ports A gelöscht und alle übrigen gesetzt. Daraufhin wiederholt sich die Kontrolle des Ports B.

Sind mehrere Tasten gedrückt, so werden entsprechend mehr Bits des Ports B gelöscht. Daraus wird ersichtlich, daß es möglich ist, mehrere Tasten gleichzeitig abzufragen.

Das folgende Programm fragt mehrere Tasten auf einmal ab. Es springt nur aus der Schleife, falls 'M', '8' und 'D' gleichzeitig gedrückt sind:

```
C000 SEI          Interrupt verhindern
C001 LDA #$00     Port A
C003 STA $DC00    auf LOW setzen
C006 LDA $DC01    Port B laden
C009 CMP #$FF     Taste gedrückt?
C00B BEQ $C001    wenn nein, dann wieder vom Anfang
C00D LDX #$00     Zähler auf Null setzen
C00F LDA #$FE     Alle Bits bis auf das erste gesetzt
```

```
C011 PHA            Wert speichern
C012 STA $DC00      an Port A übergeben
C015 LDA $DC01      Wert von Port B holen
C018 CMP $C02D,X    mit Wert aus Tabelle vergleichen
C01B BEQ $C021      wenn ja, dann weiter
C01D PLA            ansonsten Stapel rücksetzen
C01E CLC            und zum
C01F BCC $C001      Anfang springen
C021 PLA            gespeicherten Wert holen
C022 INX            Zähler erhöhen
C023 CPX #$08       Mit Endwert vergleichen
C025 BEQ $C02B      wenn ja, dann Rücksprung
C027 SEC            Carry setzen
C028 ROL            gelöschtes Bit um eins verschieben
C029 BNE $C011      unbedingter Sprung
C02B CLI            Interrupt freigeben
C02C RTS            Rücksprung
```

Tastencodes:

```
C02D FF FF FB F7 EF FF FF FF
```

Hier der dazugehörige BASIC-Loader:

```
100 FORI=1TO53STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A:NEXT:READA:IFC=ATHENC=0:NEXT:
END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,A9,00,8D,00,DC,AD,01,DC,C9,FF,F0,F4,A2,00, 98
301 DATAA9,FE,48,8D,00,DC,AD,01,DC,DD,2D,C0,F0,04,68, 8
302 DATA18,90,E0,68,E8,E0,08,F0,04,38,2A,D0,E6,58,60, 132
303 DATAFF,FF,FB,F7,EF,FF,FF,FF,00,00,00,00,00,00,00, 220
```

Die Bits des Ports A werden einzeln und der Reihe nach gelöscht und die dazugehörige Bitkombination des Ports B überprüft. Daraus ergeben sich acht Werte für Port B, die ab Adresse $C02D abgelegt sind.

Wie schon erwähnt, holt sich das Betriebssystem den ASCII-Wert der gedrückten Taste aus einer Tabelle. Zuvor wird jedoch die soeben gedrückte Taste in einem anderen Code in der Adresse $CB abgelegt. Dieser Code entspricht der Stellung des entsprechenden ASCII-Wertes in der eben erwähnten Tabelle ($EB81). Die entsprechende Tabelle zur Identifizierung der Tasten finden Sie am Ende dieses Kapitels.

Noch ein kleiner Tip: verwenden Sie bei der Tastenabfrage keine dicht aneinanderliegenden Tasten, weil die Vernetzung der Tastaturplatine schlecht ist. Es könnte passieren, daß beim Drücken nicht verzeichneter Tasten das Programm trotzdem aus der Schleife springt.

Die eigenen Tastencodes können Sie ab der Adresse $C02D eintragen. Es empfiehlt sich, nicht mehr als drei Tasten zu verwenden, weil wie schon gesagt das Programm aus der Schleife springen kann, ohne die richtigen Tasten betätigt zu haben. Beim Eintragen brauchen Sie nur die Bits der jeweiligen Taste zu löschen und einzutragen.

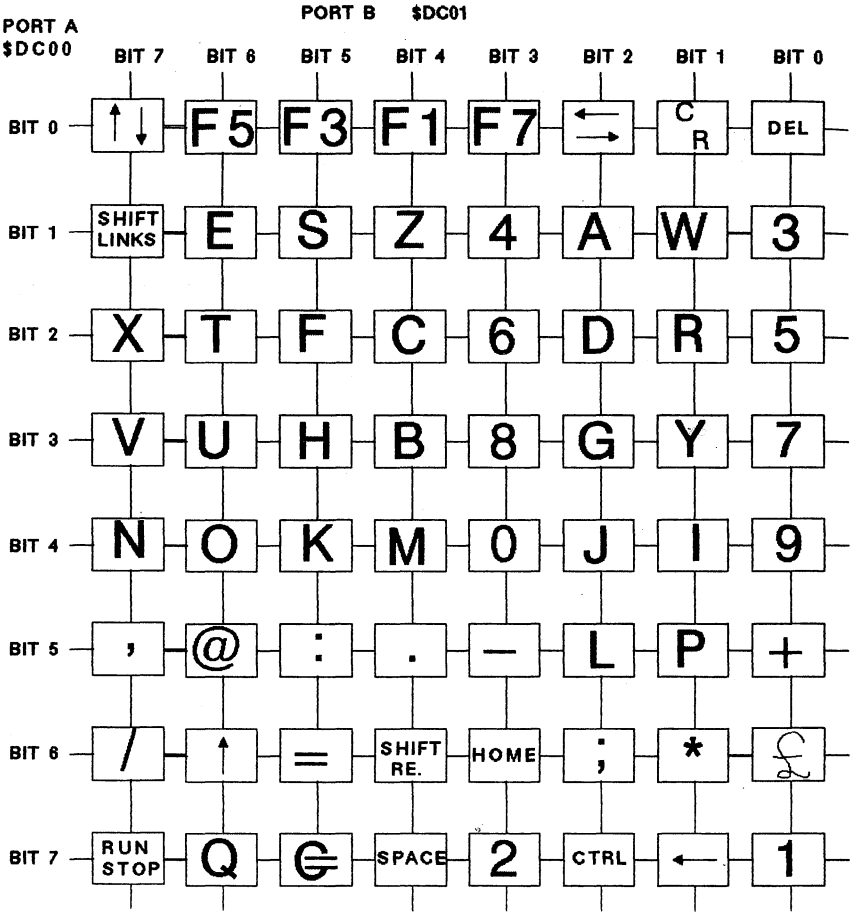


Abb. 3.6: Tastatur-Matrix

4. Directory-Schutz

4.1 Versteckter Filename

Durch Directory-Manipulation ist es möglich, einen Fremdzugriff auf Programm-Files zu verhindern. Wir möchten Ihnen hier ein paar Beispiele demonstrieren, die das Laden oder den Einblick in das Directory erschweren.

Fangen wir am besten direkt mit einem Beispiel an, mit dem Sie einen Programmnamen "verstecken" können, der dann nicht ohne weiteres geladen werden kann. Geben Sie ein:

```
SAVE"[SHIFT-SPACE]hallo",8
```

Das SHIFT-SPACE bedeutet, daß Sie zuerst die SHIFT- und dann die SPACE-Taste drücken müssen. Beide Tasten müssen also einen Augenblick gleichzeitig gedrückt sein. Anschließend können Sie den Filenamen eingeben.

Es ist aber auch möglich, direkt nach dem SHIFT-SPACE ein Hochkomma zu setzen. Wenn man das Directory listet, sieht dies dann folgendermaßen aus:

```
0 DISKETTENNAME,ID
1   ""HALLO    PRG     (Beispiel 1)
1   ""         PRG     (Beispiel 2)
662 BLOCKS FREE.

READY.
```

Wenn Sie nun versuchen, einen der beiden Programme zu laden, werden Sie immer die Fehlermeldung "MISSING FILENAME ERROR" oder "FILE NOT FOUND ERROR" erhalten. Das hat folgenden Grund: der Computer erkennt die Hochkommata als Anfangs- und Endmarkierung und ist dadurch nicht in der Lage, den dahinterliegenden Filenamen zu erfassen. Das zweite Hochkomma wird durch das abgespeicherte SHIFT-SPACE erzeugt, das den Wert $A0 hat. In dem Directory dient das $A0

als Markierung. Dem Prozessor wird nun das Ende des Filenamens "vorgegaukelt", sodaß er das zweite Hochkomma setzt. Sie werden es bestimmt als widersinnig empfinden, wenn Sie Ihre eigenen Programme nicht wieder laden können. Aber die beiden Programme sind mit demselben Trick zu laden, mit dem sie auch gespeichert wurden:

```
LOAD"[SHIFT-SPACE]HALLO",8
```

oder

```
LOAD"[SHIFT-SPACE]",8
```

Mann kann dadurch zum Beispiel auch ein "verstecktes" Programm über ein Ladeprogramm aufrufen, um so das Hauptprogramm zu schützen.

4.2 Verstecken der Filenamen durch Steuerzeichen

Auch die Steuerzeichen eignen sich durchaus für den Directory-Schutz. Unter Steuerzeichen versteht man die CURSOR-Tasten, die Farbtasten, die Delete- und die Insert-Taste. Die Steuerzeichen sind nur im Hochkomma-Modus sichtbar und können daher nicht gelistet werden, sondern werden während eines LIST-Vorgangs ausgeführt. Auf diese kleine Schwäche der LIST-Funktion stützt sich auch unser Schutzverfahren. Geben Sie dazu folgende Zeile ein:

```
SAVE""[DEL]HAL[INS][DEL]LO",8
```

Das ganze nochmal in Worten: Nach dem ersten Hochkomma setzen Sie noch ein zweites, um den Hochkomma-Modus abzuschalten. Anschließend drücken Sie die DEL-Taste, um das zweite Hochkomma zu löschen, und geben die ersten drei Buchstaben des Filenamens ein. Danach drücken Sie zuerst die INS- und anschließend die DEL-Taste, um ein reverses "T" zu erzeugen. Jetzt kann man den Rest des Filenamens eingeben und abspeichern. Dies sieht im Directory folgendermaßen aus:

```
0 DISKETTENNAME,ID
1   HALO       PRG
663 BLOCKS FREE.

READY.
```

Es wird Ihnen bestimmt aufgefallen sein, daß ein Buchstabe des Filenamens verschwunden ist, und zwar ein "L", obwohl der Filename mit zwei "LL" abgespeichert wurde. Daran ist, wie schon erwähnt, unser Steuerzeichen schuld. Wenn man nun versucht, das Programm auf normale Art und Weise zu laden, meldet der Computer wieder "FILE NOT FOUND ERROR", weil ihm das zweite "L" fehlt, um das Programm im Directory zu finden. Das Laden des Programms erfolgt nach dem gleichen Prinzip, wie es abgespeichert wurde:

```
LOAD"" [DEL] HAL [INS] [DEL] LO",8
```

Mit den Farbtasten kann man auch einen bunten Filenamen erzeugen, wenn man das Programm mit den jeweiligen Farb-Codes abspeichert. Zum Beispiel so:

```
SAVE"H [CTRL 1] A [CTRL 2] L [CTRL 3] L [CTRL 4] O",8
```

Mit CTRL 1 bis CTRL 4 sind die Farb-Codes auf der Tastatur gemeint. Auch diese Codes können nicht gelistet werden, sie verändern nur die Farbe der Buchstaben. Auch diesmal muß das Programm genau so geladen werden, wie es gespeichert wurde.

Noch ein wichtiger Hinweis: Dieser LIST-oder DIRECTORY-Schutz funktioniert nur auf dem originalen Betriebssystem. Wenn Sie zufällig Speed-DOS oder ein anderes Betriebssystem verwenden, ist es durchaus möglich, daß dieser Schutz nicht funktioniert, weil eine andere LIST-Routine verwendet wird.

4.3 Der "blinde" Block

Vor der Praxis möchten wir Ihnen den Aufbau des Directorys verdeutlichen. Der Directory-Block beginnt, falls keine Mani-

pulationen vorgenommen wurden, immer ab Track $12 und Sektor $01. Wenn wir uns diesen Block einmal mit dem Diskmonitor ansehen, sehen wir folgendes:

```
00: 12 04 82 11 00 48 41 4C .....HAL
08: 4C 4F 31 A0 A0 A0 A0 A0 LO1.....
10: A0 A0 A0 A0 A0 00 00 00 ........
18: 00 00 00 00 00 00 01 00 ........
20: 00 00 83 11 01 48 41 4C .....HAL
28: 4C 4F 32 A0 A0 A0 A0 A0 LO2.....
30: A0 A0 A0 A0 A0 00 00 00 ........
38: 00 00 00 00 00 00 01 00 ........
```

Die ersten beiden Bytes in Zeile $00 stellen den Block-Linker dar. Dieser hat die Aufgabe, gegebenenfalls auf den nächsten Directory-Block zu zeigen. Die erste Zahl zeigt auf den Track und die andere auf den Sektor, meistens $12 $04(18 01). Das dritte Byte bestimmt den Filetyp ($81=SEQ, $82=PRG, $83=USR, 84=REL).

Die nächsten beiden Bytes zeigen auf den Track und Sektor des Programms mit dem Namen "HALLO1". Anschließend folgt der Filename, für den genau 16 Bytes reserviert sind.

Nach den ASCII-Codes folgen noch einige $A0, die sowohl als Platzhalter als auch als Endmarkierung bestimmt sind, damit der Prozessor weiß, wann er das zweite Hochkomma setzen muß. Er setzt es automatisch, wenn er nach den ASCII-Codes auf ein $A0 trifft.

Die letzten beiden Bytes in Zeile $18 stellen in LOW- und HIGH-Byte die Block-Anzahl des Programms dar. In unserem Fall ist das Programm einen Block lang. Nach dem Sie jetzt über den Aufbau des Directorys Bescheid wissen, können wir mit der Praxis beginnen.

Im Prinzip ist dieser Schutz einfach aufgebaut, jedoch äußerst wirkungsvoll. Er macht das Listen des Directorys schier unmöglich. Man muß nur in den ersten beiden Bytes in Zeile $00 "FF" hineinschreiben. Die Floppy wird bei einem Leseversuch des

Directorys mit der Fehlermeldung "ILLEGAL TRACK OR SECTOR" antworten, weil der Block-Linker auf einen nicht existenten Block zeigt.

Jedoch können die Programme weiterhin normal geladen werden, es sei denn, das Directory geht über $12 $01 hinaus. Dann muß der Block-Linker nicht in $12 $01, sondern in $12 $04 usw. geändert werden, weil sonst die untersten Programme im Directory von der Floppy nicht gefunden werden können.

Falls Sie jedoch keinen Maschinensprachemonitor zur Hand haben, können Sie auch das folgende BASIC-Programm eingeben, das für Sie die Manipulation vornimmt:

```
10 OPEN 1,8,2,CHR$(0)+"P,R":CLOSE 1
20 OPEN 1,8,15:OPEN 2,8,2,"#"
30 PRINT#1,"M-R"CHR$(24)CHR$(0)CHR$(2)
40 GET#1,A$,B$
50 T$=STR$(ASC(A$+CHR$(0)))
60 S$=STR$(ASC(B$+CHR$(0)))
70 PRINT#1,"U1 2 0"+T$+S$
80 PRINT#1,"B-P 2 0"
90 PRINT#2,CHR$(255);
100 PRINT#1,"U2 2 0"+T$+S$
110 CLOSE 1:CLOSE 2
```

Die Zeilen 10 bis 60 haben die Aufgabe, den letzten Directory-Block zu finden. Das geht folgendermaßen vor sich: In Zeile 10 wird ein File eröffnet, das nicht auf Diskette vorhanden ist, damit das Betriebssystem der Floppy das ganze Directory nach diesem File durchsucht. Das Betriebssystem wird das File natürlich nicht finden, was auch der Zweck dieser Operation ist. Dadurch haben wir aber direkt den letzten Directory-Block gefunden. Man muß ihn anschließend nur aus der Floppy auslesen. Diese Operation übernehmen die Programmzeilen 20 bis 60. Nachdem dieses geschehen ist, braucht man nur diesen Block in den Puffer der Floppy einzulesen und das entsprechende Byte zu

ändern. Dies passiert in den Zeilen 70 bis 90. Die letzten beiden Zeilen haben dann nur noch die Aufgabe, den Block wieder auf Diskette zu speichern und die Kanäle zu schließen.

Falls Sie einmal die Absicht haben, den Schutz wieder zu entfernen, müssen Sie nur in der Zeile 90 statt CHR$(255); ein CHR$(0); eintragen. Nach dem Programmstart ist der Schutz deaktiviert, und das Directory kann dann wieder normal gelistet werden.

Dazu muß wieder gesagt werden, daß dieser Schutz nur auf dem originalen Betriebssystem einwandfrei funktioniert, weil das Directory nicht ins BASIC geladen, sondern sofort auf den Bildschirm gebracht wird.

4.4 Der Trick mit den Nullen

Bei diesem Schutz manipulieren wir nicht das eigentliche Directory, sondern den Diskettennamen. Der Diskettenname ist in der BAM, die immer auf Track $12 und Sektor $00 liegt, abgelegt. Mit dem Diskettenmonitor sieht dies dann folgendermaßen aus:

```
00: 12 01 41 00 15 FF FF 1F  ..A.....
08: 15 FF FF 1F 15 FF FF 1F  ........
.
.
.
90: 44 49 53 4B A0 A0 A0 A0  DISK....
98: A0 A0 A0 A0 A0 A0 A0 A0  ........
A0: A0 A0 30 38 A0 32 41 A0  ..08.2A.
A8: A0 A0 A0 00 00 00 00 00  ........
B0: 00 00 00 00 00 00 00 00  ........
.
.
.
```

Der Diskettenname in Zeile $90 ist einfach in ASCII-Codes abgelegt. Er ist ähnlich wie ein Filename im Directory aufgebaut. Hinter den ASCII-Codes folgen wieder die 16 Platzhalter

$A0. Sofort dahinter folgen die ID und das Formatkennzeichen. Die restlichen Bytes sind unbenutzt.

Auch dieser Schutz ist nicht schwer zu installieren. Wir überschreiben einfach in Zeile $90 die ersten 6 Bytes des Diskettennamens mit den Werten "14 14 14" und "00 00 00". Die Zeile $90 sieht dann so aus:

```
90: 14 14 14 00 00 00 A0 A0  ........
```

Speichern Sie den Block wieder auf $12 $00 und gehen Sie mit "X" wieder in BASIC. Dort initialisieren Sie das Laufwerk mit OPEN 1,8,15,"I", damit sich die Floppy die neue BAM in den Puffer holt. Normalerweise geschieht dieser Vorgang bei jedem Diskettenwechsel automatisch, vorausgesetzt, die IDs sind verschieden.

Falls Sie gerade keinen Maschinensprachemonitor zur Hand haben, nimmt Ihnen dieses kleine BASIC-Programm die Arbeit ab:

```
10 OPEN 3,8,3,"#"
20 OPEN 15,8,15
30 PRINT#15,"U1 3 0 18 0"
40 PRINT#15,"B-P 3 144"
50 PRINT#3,CHR$(20)CHR$(20)CHR$(20)CHR$(0)CHR$(0)CHR$(0)CHR$(160);
60 PRINT#15,"U2 3 0 18 0"
70 PRINT#15,"I"
80 CLOSE 3:CLOSE 15
```

Wenn Sie jetzt versuchen, das Directory mit LOAD"$",8 zu laden, erscheint sofort nach dem LIST die READY-Meldung. Die Programme lassen sich aber immer noch ganz normal laden.

Das Directory ist ganz normal in den BASIC-Speicher geladen worden, aber es läßt sich nicht listen. Schuld daran sind die drei Nullen. Der BASIC-Interpreter setzt diese Nullen als End-Markierung ans Ende eines BASIC-Programms. Stößt nun der

BASIC-Interpreter während des LIST-Vorgangs auf drei Nullen, erkennt er sie als End-Markierung und beendet den LIST-Vorgang mit einer READY-Meldung.

Unser Directory-Schutz beruht auf dem gleichen Prinzip. Das Directory wird samt Diskettenname in den BASIC-Speicher gelesen. Da aber in dem Diskettennamen nur "14 14 14 00 00 00" steht, kann das Directory nicht gelistet werden. Der Wert $14 ist nichts weiter als das Steuerzeichen DEL. Dadurch werden die ersten drei Bytes (0 "), die der Floppy-Prozessor automatisch setzt, sofort wieder gelöscht. Die danach folgenden Nullen setzen die BASIC-Endmarkierung, wodurch der weitere LIST-Vorgang automatisch beendet wird. Auf dem Bildschirm bleibt nur noch die READY-Meldung sichtbar.

Auch dieser Schutz funktioniert nur auf dem originalen Betriebssystem, weil das Directory bei Speed-DOS oder anderen Betriebssystemen nicht ins BASIC geladen, sondern direkt in den Bildschirmspeicher geschrieben wird. Dies hat den Vorteil, daß das bereits im Speicher vorhandene BASIC-Programm nicht überschrieben werden kann.

4.5 Das geteilte Directory

Mit dem gleichen Verfahren ist es möglich, ein Directory zu "halbieren". Das heißt, der obere Teil ist sichtbar und der Rest ist unsichtbar, also nicht zu listen. Das sieht so aus:

```
0"DISKETTENNAME",ID
1   "LOADER"    PRG
200BLOCKS FREE.

READY.
```

Folglich bleiben die Programme, die nach dem LOADER folgen, unsichtbar.

Die "BLOCKS FREE"-Meldung ist nichts weiter als ein manipulierter Filename, hinter dem noch drei Nullen stehen, also die BASIC-Endmarkierung. Sehen wir uns das doch einmal im Diskmonitor an:

```
00: 12 04 82 11 00 4C 4F 41   .....LOA
08: 44 45 52 A0 A0 A0 A0 A0   DER.....
10: A0 A0 A0 A0 A0 00 00 00   ........
18: 00 00 00 00 00 00 01 00   ........
20: 00 00 82 11 01 14 14 42   .......B
28: 4C 4F 43 4B 53 20 46 52   LOCKS FR
30: 45 45 00 00 00 00 00 00   EE......
30: 00 00 00 00 00 00 FF 00   ........
```

Man sieht in Zeile $20 eindeutig, daß die "BLOCKS FREE"-Meldung ein ganz normales File ist. Aber vor den eigentlichen ASCII-Zeichen sind noch zwei Bytes mit dem Wert $14 eingefügt. Es sind die Steuerzeichen für DEL (Backspace), die dafür Sorge tragen, daß das Hochkomma gelöscht wird und daß die ASCII-Zeichen näher an den Rand gerückt werden, damit das Ganze realistischer aussieht. Nach den ASCII-Zeichen folgen noch die drei Nullen, die jegliches Listen verhindern. So bleiben die dahinterliegenden Programme unsichtbar.

Das Ganze läßt sich auch vom BASIC aus realisieren. Als Voraussetzung muß die Diskette, auf der der Schutz aufgetragen werden soll, leer sein. Jetzt kann man den LOADER auf Diskette abspeichern. Wenn man keinen LOADER haben möchte, kann man auch direkt den manipulierten Filenamen auf Diskette aufbringen. Dieses funktioniert dann so:

```
SAVE CHR$(20)+CHR$(20)+"BLOCKS FREE"+CHR$(0)+CHR$(0)+CHR$(0)
```

Die weiteren Programme, die nun auf diese Diskette gespeichert werden, werden hinter diesem Filenamen gespeichert und bleiben deshalb unsichtbar. Auch hierzu ist zu sagen, daß dieser Schutz nur im Original-Betriebssystem funktioniert. Mit dem Speed-DOS-Betriebssystem ist auch dieser List- oder Directory-Schutz unwirksam.

4.6 Die Tarnung des Filetyps

Eine äußerst wirkungsvolle Methode zum Schutz eines Programms vor Fremdzugriffen ist auch das Ändern eines Programms in eine sequentielle Datei. Das läßt sich mit einem Diskmonitor ohne weiteres realisieren. Man braucht nur ein Byte zu ändern:

```
00: 00 FF 81 11 00 4D 41 49   .....MAI
08: 4E A0 A0 A0 A0 A0 A0 A0   N.......
10: A0 A0 A0 00 00 00 00 00   ........
18: 00 00 00 00 00 00 20 00   ........
```

Man muß nur beim dritten Byte in Zeile $00 eine $81 hineinschreiben. Nach dem Abspeichern des Blocks sieht das Directory dann wie folgt aus:

```
0"DISKETTENNAME",ID
32   "MAIN"      SEQ
255 BLOCKS FREE
```

Falls es sich um ein BASIC-Programm handelt, kann man es auch mit SAVE"MAIN,S,W",8 abspeichern. Das hat dann die gleiche Wirkung. Mit einem normalem LOAD"MAIN",8 läßt sich das Programm nicht laden, weil das Betriebssystem mit Hilfe dieses Wertes den Filetyp feststellt und einen Lade-Vorgang nicht zuläßt. Aber mit einem kleinem Trick läßt sich auch dieses Programm laden.

```
LOAD "MAIN,S,R",8
```

Auf diese Weise teilt man dem Betriebssystem mit, daß man ein sequentielles File öffnen möchte. Anschließend läßt sich das Programm ohne Schwierigkeiten in den Speicher laden.

Noch ein kleiner Trick: man kann auf diese Weise ein Lade-Programm in BASIC schreiben und dieses LOADER nennen. Das Hauptprogramm "MAIN", das im Directory als sequentielles File erscheint, wird vom LOADER nachgeladen, wenn ein bestimm-

tes Password eingegeben wurde. Auf diese Weise läßt sich auch ein Programm vor Fremdbenutzung schützen. Diese Methode läßt sich bei allen Programm-Files durchführen, so daß Sie auf diese Weise alle wichtigen Programme sichern können.

4.7 Gelöschte Files

Als nächstes wollen wir einzelne Files kurzzeitig "wiederbeleben". Dazu brauchen wir die Blocklese- und Schreibbefehle. Außerdem benötigen wir eine leere Diskette. Dann speichern wir das zu sichernde Programm und anschließend folgendes Ladeprogramm ab:

```
10 OPEN 3,8,3,"#"
20 OPEN 15,8,15
30 PRINT#15,"U1 3 0 18 1"
40 PRINT#15,"B-P 3 2"
50 PRINT#3,CHR$(130);
60 PRINT#15,"U2 3 0 18 1"
70 PRINT#15,"I"
80 CLOSE 15
90 CLOSE 3
100 LOAD"PROGRAMMNAME",8
```

Wichtig ist, daß diese Reihenfolge streng eingehalten wird. Und nun benutzen wir den Scratch-Befehl der Floppy, um das erste File, also das Hauptprogramm, wieder zu löschen. Achtung! Ab sofort dürfen keine Schreibzugriffe auf diese Diskette mehr ausgeführt werden.

Im Directory ist jetzt nur das zweite File sichtbar. Das erste File steht immer noch im Directory, allerdings mit dem Filetyp *DEL. Der Stern vor dem DEL bedeutet, daß ein nicht ordnungsgemäß geschlossenes File vorliegt. Solche Sternchen findet man, wenn das Abspeichern eines Programms auf Diskette wegen Platzmangels nicht korrekt abgeschlossen werden kann.

Da sich die Floppy als "intelligent" bezeichnet, unterschlägt sie die *DEL-Einträge beim Laden von Directories in den Computer-Speicher, um die gelöschten Files nicht zu listen. Natürlich ist das File auch noch auf der Diskette vorhanden, es ist für uns nur nicht mehr sichtbar.

Um es wieder zu regenerieren, muß das für den Filetyp zuständige Byte des Eintrags auf den Wert $82 (130) gesetzt werden. Beim ersten Eintrag ist das Byte Nummer 2, also das dritte (Zählungen beginnen bei Computern grundsätzlich bei Null) im Block Track 18 Sektor 1.

Wichtig ist, daß die Diskette vor dem Laden des Hauptprogramms initialisiert wird. Die Floppy merkt sich nämlich immer einige Teile des Directories und liest diese erst gar nicht von der Diskette, wenn ein Directory abgerufen wird. Die Folge wäre, daß wir immer noch mit dem alten Directory arbeiten, obwohl das neue auf der Diskette steht. Beim Initialisieren werden dann allerdings die internen Zwischenspeicher des Laufwerks aktualisiert.

Die ersten Befehle im Hauptprogramm sollten dieses wieder "löschen", indem die obere Sequenz verwendet wird, mit dem Unterschied, daß anstelle des CHR$(130) ein CHR$(0) steht. Dieses ist die Kennzeichnung für ein *DEL-File. Das geht schneller und weniger auffällig als ein Scratch-Befehl. Dieses Prinzip läßt sich fast beliebig ausweiten, so zum Beispiel, wenn man die Filenamen verändert oder den falschen Programm-Linker wieder richtig stellt.

Die beste Methode: der Compiler-Schutz

Zum Abschluß dieses Kapitels noch einige allgemeine Tips: Schutz-Programme, die in BASIC geschrieben wurden, sollte man unbedingt compilieren. Jeder Compiler erstellt einen derartig verwirrten, fast unlesbaren Code, daß selbst die versiertesten Knack-Profis resignieren müssen.

Wichtig ist aber, daß Befehle, die an die Floppy gesendet werden, im Programmtext verschlüsselt stehen sollten. Zum Beispiel, indem sie mit dem CHR$-Befehl arbeiten und die einzusendenden Zahlenwerte über einen Entschlüsselungsalgorithmus errechnet werden. Dann kann man dem Compilat nicht einmal über einen Monitor entnehmen, welche Befehle zur Floppy gesendet werden.

5. Kopierschutz mit der Datasette

Da viele C64-Besitzer immer noch mit einer Datasette arbeiten und es für etliche Firmen billiger ist, ihre Programme auf Kassetten anstatt auf Disketten zu vertreiben, darf in diesem Buch ein Kapitel über Kassettenkopierschutz nicht fehlen.

5.1 Sinn und Unsinn eines Kassettenkopierschutzes

Ganz am Anfang dieses Kapitels muß die Frage stehen, ob es überhaupt sinnvoll ist, ein auf Band aufgenommenes Programm vor dem Kopieren schützen zu wollen. Für einen Kopierer ist es nämlich ein leichtes, zwei Kassettenrecorder mit Hilfe eines Überspielkabels miteinander zu verbinden, um sich dann ein Duplikat des Programms anzufertigen. Dieses Verfahren ist jedoch zu unrationell, als daß ein auf diese Weise kopiertes Programm auf dem schwarzen Markt weite Verbreitung finden könnte. Schließlich wird jede weitere Kopie, die man von einer vorhandenen Kopie anlegt, in der Qualität etwas schlechter sein als der Vorgänger, was dazu führt, daß sich ab einer gewissen Stufe das Programm nicht mehr einladen läßt. Außerdem müssen zur Anfertigung eines Duplikats sowohl Aussteuerung als auch Tonkopfeinstellung des Kassettenrecorders exakt eingestellt sein.

Ein weiteres Argument, das gegen das Erstellen einer 1:1-Kopie spricht, ist der Platzbedarf. Man kann unter Verwendung von Turbo-Tape auf einer 60-Minuten-Kassette etwa 40 bis 50 raubkopierte Programme mittlerer Länge unterbringen, bei denen der Kopierschutz entfernt wurde. Wenn jedoch der Kopierschutz noch vorhanden ist, also die Kopie mit Hilfe zweier Recorder angefertigt wurde, reicht dasselbe Band maximal für ca. acht bis neun Programme. Die meisten Raubkopierer sind sowieso an einer wesentlich schneller lad- und kopierbaren Diskettenversion eines Programms interessiert.

Wer Wert darauf legt, daß ein von ihm entworfenes Programm auch nicht ein einziges Mal kopiert wird, ist mit einer Bandaufnahme seines Programms völlig hilflos. Wer aber sein

Programm vor einer schnellen und ausgedehnten Verbreitung auf dem schwarzen Markt schützen will, für den bietet die Datasette mehr Kopierschutzmöglichkeiten als eine Diskettenstation. Denn für das Diskettenlaufwerk existieren im Gegensatz zur Datasette unzählige von Kopierprogrammen, die von etwa 90% der geschützten Software funktionsfähige Kopien erstellen können. Jede Kopie steht dann dem Original um nichts in der Qualität nach und läßt sich auch ebenso schnell weitervervielfältigen.

Solche Kopierprogramme für die Datasette zu erstellen, ist aus Gründen, auf die wir später noch genauer eingehen werden, nicht möglich. Existierende Kassettenkopierprogramme erlauben es höchstens, das vom Betriebssystem verwendete Aufzeichnungsformat oder das in Deutschland nahezu zum Standard gewordene Turbo-Tape-Format zu kopieren. Da aber die meisten Softwarefirmen, die ihre Programme auf Kassetten vertreiben, ein eigenes Aufzeichnungsformat verwenden, sind solche Kopierprogramme zum Kopieren von Originalen untauglich.

Einige Raubkopierer sind soweit gegangen, speziell für einige von Firmen wiederholt verwendete Formate Kopierprogramme zu schreiben. Prinzipiell braucht man aber für jedes neue Aufzeichnungsverfahren ein neues "Copy". Ein generelles Kopierprogramm existiert bis zum jetzigen Zeitpunkt noch nicht, und es hat auch den Anschein, daß es unmöglich ist, jemals ein solches Programm zu entwerfen.

5.2 Laden und Abspeichern von Programmen und Daten

5.2.1 Was man von BASIC aus machen kann

Wer einen Kopierschutz entwerfen will, sollte zuerst einmal wissen, auf welche verschiedenen Arten man überhaupt etwas auf einer Kassette abspeichern und wieder einladen kann. Das Betriebssystem des C64 stellt uns dazu prinzipiell zwei Möglichkeiten zur Verfügung.

Die erste und einfachste Möglichkeit ist die, mit den Befehlen LOAD und SAVE zu arbeiten.

```
SAVE "Programmname" oder SAVE "Programmname",1
```

speichert das im Computer befindliche BASIC-Programm auf Band,

```
LOAD "Programmname" oder LOAD "Programmname",1
```

lädt ein Programm vom Band wieder zurück in den Rechner. Sie als Anwender brauchen sich bei dieser Prozedur nicht im geringsten darum zu kümmern, wie und wo Ihr Programm im Rechnerspeicher untergebracht wird. Diese Arbeit erledigt für Sie der BASIC-Interpreter.

Bei dieser Aufzeichnungsmethode werden auf dem Band zwei unterschiedliche Teile abgelegt. Der erste Teil ist der sogenannte 'Header'. Er enthält neben dem Programmnamen auch noch Daten über die Lage des Programms im Speicher und seine Länge. Der zweite Teil nun enthält das eigentliche Programm. Diese Unterteilung wird auch beim Einladen des Programms sichtbar, wenn der Computer nach Einlesen des Headers sich erst noch einmal zur Anzeige des Programmnamens meldet. Danach wird erst das eigentliche Programm vom Band geholt.

Das zweite Verfahren, mit der Datasette Daten aufzuzeichnen, liefert der OPEN-Befehl. Damit ist es möglich, vom laufenden BASIC-Programm aus Zahlen, Buchstaben oder Variablenwerte auf Band zu sichern und von dort auch wieder zurückzuholen. Die Befehlsfolgen dazu sehen folgendermaßen aus:

```
OPEN 1,1,1,"Filename"
```

eröffnet ein sogenanntes 'Datenfile' auf der Kassette, das heißt, man kann jetzt Daten beliebig hintereinander auf dem Band ablegen. Dies geschieht mit Hilfe des Befehls

```
PRINT# 1, ... (Variablen,Daten wie bei PRINT).
```

Dieser Befehl funktioniert genau wie der PRINT-Befehl, nur daß die Werte nicht auf dem Bildschirm, sondern zur Datasette hin ausgegeben werden.

Der dritte Befehl, den man noch benötigt, um seine Daten abzuspeichern, teilt dem Computer mit, daß die Aufzeichnung beendet werden soll. Er lautet:

```
CLOSE 1
```

Wenn man diesen Befehl vergißt, kann es dazu kommen, daß einem hinterher Daten auf dem Band fehlen.

Was bedeuten nun die Zahlen, die den Befehlen mitgegeben werden müssen? Die erste Eins hinter dem Befehl OPEN ist die 'logische Filenummer'. Sie kann einen beliebigen Wert von 0 bis 255 annehmen. Es ist dieselbe Zahl wie hinter dem PRINT#- und CLOSE-Befehl, denn ihr Zweck ist es, diesen beiden Befehlen mitzuteilen, auf welchen OPEN-Befehl sie sich beziehen. Die zweite Zahl heißt 'Geräteadresse', da sie angibt, welches Gerät (hier die Datasette) der OPEN-Befehl ansprechen soll. Die Eins hinter SAVE und LOAD hat genau dieselbe Funktion.

Die dritte Zahl, die sogenannte 'Sekundäradresse', ist die interessanteste, da sie je nach angesprochenem Gerät unterschiedliche Funktionen erfüllt. Bei der Datasette hat eine Eins die Bedeutung 'Datenfile zum Schreiben eröffnen'. Um nämlich die geschriebenen Daten wieder lesen zu können, braucht man bloß ein Datenfile mit der Sekundäradresse Null zu eröffnen:

```
OPEN 1,1,0,"Filename"
```

Das Einlesen der Daten erfolgt dann mit den Befehlen

```
GET#1,... (Folge von Variablen)
```

oder

```
INPUT#1,... (Folge von Variablen).
```

Diese beiden Anweisungen funktionieren genauso wie ihre Gegenstücke GET und INPUT, nur werden die Zeichen nicht über die Tastatur eingetippt, sondern vom Band eingelesen. Auch hier sollte man nach Abschluß des Einlesens mit CLOSE 1 den Datenkanal wieder schließen.

Nun zur Praxis. Folgendes Beispielprogramm liest eine vorher bekannte Anzahl von Zahlen von der Tastatur und schreibt sie auf Band:

```
10 OPEN 1,1,1,"DATEI":REM Datei zum Schreiben eröffnen
20 PRINT"WIEVIELE ZAHLEN WOLLEN SIE ABSPEICHERN"
30 INPUT A:PRINT#1,A:REM Anzahl der Daten auf Band
40 FOR I=1 TO A
50 PRINT I;". ZAHL";:INPUT B
60 PRINT#1,B:REM eine Zahl abspeichern
70 NEXT I
80 CLOSE 1:REM File schließen
```

Das dazugehörige Programm, das die Daten wieder einliest und anzeigt, sieht so aus:

```
10 OPEN 1,1,0,"DATEI":REM Datei zum Lesen öffnen
20 INPUT#1,A:REM Datenanzahl einlesen
30 FOR I=1 TO A
40 INPUT#1,B:REM eine Zahl einlesen
50 PRINT I;". ZAHL:";B
60 NEXT I
70 CLOSE 1
```

Mit diesem Handwerkszeug können wir nun endlich zum ersten Kopierschutz übergehen. Er besteht einfach darin, daß man hinter sein Programm ein Datenfile auf das Band schreibt, die Daten dann einliest und vergleicht, ob sie stimmen. Wenn nicht, so läuft auch das Programm nicht.

Angenommen, man hat mit dem oben angegebenen Beispielprogramm die vier Zahlen 3, 4.1, -10 und 1 auf Band abgelegt, dann könnte das Abfrageprogramm wie folgt aussehen:

```
10 OPEN 1,1,0,"DATEI"
20 INPUT#1,A,B,C,D,E:REM 5 Werte (mit Datenanzahl) lesen
30 CLOSE1
40 IF A<>4 OR B<>3 OR C<>4.1 OR D<>-10 OR E<>1 THEN PRINT"LAUEFT NICHT":
STOP
50 PRINT"KOPIERSCHUTZ ERKANNT"
```

Wer so etwas kopieren will, kommt mit LOAD und SAVE nicht weiter. Er muß sich ein Programm dafür schreiben. Zugegeben, dies ist nicht besonders kompliziert. Man kann aber hier ein Handikap einbauen, das es schwer macht zu erkennen, wieviele Daten überhaupt kopiert werden müssen. Ein Kopierprogramm für Datenfiles müßte nämlich normalerweise so aussehen, daß es jedes einzelne Daten-Byte mit GET# in ein Array einliest und dann anhand der Status-Variablen ST nachprüft, ob schon alle Daten gelesen wurden. Diese Variable erhält nämlich den Wert 64, sobald man das letzte Daten-Byte geholt hat. Das Betriebssystem erkennt das Ende des Datenfiles daran, daß es hinter das File das Zeichen 'CHR$(0)' schreibt. Nichts hindert uns aber, dieses Zeichen selbst auf das Band zu schreiben. Dadurch erhält ST den Wert 64 schon, bevor alle Daten eingelesen wurden. Wenn also das Kopierprogramm im Prinzip so aussieht:

```
10 DIM D$(1000):REM maximal 1000 Bytes
20 INPUT"NAME DES DATENFILES";N$
30 OPEN 1,1,0,N$
40 I=0
50 GET#1,D$(I):REM Ein Byte lesen
60 IF ST=0 THEN I=I+1:GOTO 50:REM alle Daten gelesen?
70 CLOSE 1
80 INPUT"KASSETTE WECHSELN UND RETURN DRUECKEN";A$
90 OPEN 1,1,1,N$
100 FOR J=0 TO I
110 PRINT#1,D$(J);:REM Daten wieder schreiben
120 NEXT J
130 CLOSE 1
```

dann kann man sich durch folgende Bandaufzeichnung davor schützen:

```
10 OPEN 1,1,1,"FILENAME"
20 PRINT#1,"ERSTER TEIL"
30 PRINT#1,CHR$(0);:REM Datenende vortäuschen
40 PRINT#1,"ZWEITER TEIL"
50 CLOSE 1
```

Die Abfrage dazu sieht dann folgendermaßen aus:

```
10 OPEN 1,1,0,"FILENAME"
20 INPUT#1,A$
30 GET#1,B$:REM CHR$(0) überspringen
40 INPUT#1,B$
50 CLOSE 1
60 IF A$<>"ERSTER TEIL" OR B$<>"ZWEITER TEIL" THEN STOP
70 PRINT"KOPIERSCHUTZ ERKANNT"
```

Jemand, der ein solches Datenfile duplizieren will, muß genau wissen, wieviele Daten-Bytes das File enthält. Den einzigen Hinweis, den er dazu erhält, ist, daß beim letzten Byte ST den Wert 64 annimmt, was aber nicht notwendigerweise bedeuten muß, daß das File wirklich zu Ende ist. Sie werden sich fragen, ob es möglich ist, Daten zu schreiben, ohne daß sie mit CHR$(0) beendet werden. In der Tat geht auch das. Wir werden aber erst später darauf zurückkommen.

Bei der gerade besprochenen Aufzeichnungsmethode werden die Daten in mehreren Teilen abgelegt. Als erstes kommt wie bei mit SAVE abgespeicherten Programmen ein Header mit dem Filenamen und der Information, daß es sich um ein Datenfile handelt. Danach folgen in kurzen Blöcken die Daten. Jeder Block wird zuerst in einen speziell dafür reservierten Speicherbereich, dem sogenannten Kassettenpuffer, gelesen und von da aus an das BASIC-Programm weitergeleitet. Immer wenn die Daten aus dem Kassettenpuffer vollständig übergeben wurden, wird der nächste Datenblock gesucht und eingeladen. Das macht

es einem Kopierer natürlich wieder einfacher, da er die Datenanzahl nicht aufs Byte genau festzustellen braucht, sondern nur wissen muß, wieviele Datenblöcke zu dem File gehören. Wenn er das einmal herausgefunden hat, braucht er nur alle diese Blöcke vollständig zu kopieren, ohne sich um CHR$(0)-Bytes zu kümmern. Damit er das aber nicht herausfindet, sollte man die Kopierschutzabfrage nicht für jedermann offen einsehbar im Programm unterbringen. Anregungen zu diesem Thema finden Sie im Kapitel "Programmschutz".

Sekundäradressen können noch andere Zwecke erfüllen als die oben beschriebenen. In Verbindung mit dem OPEN-Befehl ist noch die Sekundäradresse Zwei von Bedeutung. Mit

```
OPEN 1,1,2,"Filename"
```

wird ein Datenfile genauso zum Schreiben eröffnet wie unter Verwendung der Sekundäradresse Eins. Hinter das Datenfile wird dann eine EOT (= End of Tape = Bandende)-Markierung geschrieben. Diese Markierung bewirkt, daß jeder Versuch, über sie hinweg nach einem Programm oder Datenfile zu suchen, mit der Fehlermeldung 'DEVICE NOT PRESENT' abgebrochen wird. Sie dient dazu, den Punkt auf dem Band anzuzeigen, hinter dem sich nichts mehr an Aufzeichnungen befindet. Von BASIC aus läßt sich allerdings die Existenz dieser Markierung nicht ohne weiteres abfragen, da sie zum Abbruch des laufenden Programmes führt. Man kann sie höchstens überspringen, indem man über die File-Ende-Markierung hinweg mehr Daten einliest, als man aufgezeichnet hat. Es müssen nur soviele Daten angefordert werden, daß der nächste Block nach dem Datenfile, also der EOT-Block, nachgeladen wird. Als Kopierschutz ist diese Markierung allerdings kaum zu verwenden, da sie zu leicht erkannt und noch leichter wieder erzeugt werden kann.

Auch bei den Befehlen LOAD und SAVE können Sekundäradressen angegeben werden. Um den Sinn der Funktionen der Sekundäradressen zu verstehen, sollte man wissen, daß sich der Speicherbereich, in dem ein BASIC-Programm im Rechner untergebracht wird, verlegen läßt. Die Anfangsadresse des BASIC-Speichers befindet sich in den Speicherstellen 43 und 44

im LOW-HIGH-Byte Format und hat normalerweise den Wert 2049 (=$0801). Will man beispielsweise den BASIC-Start nach 3000 (=$0BB8=11*256+184) verschieben, so braucht man bloß diese beiden Adressen zu verändern. Außerdem muß nach 2999 eine Null stehen. Danach fehlt nur noch die Eingabe von NEW, und schon beginnt der BASIC-Speicher bei 3000. Die komplette Anweisung lautet also:

```
POKE 43,184:POKE 44,11:POKE 2999,0:NEW
```

Hat man ein BASIC-Programm, das an der Speicherstelle 2049 begann, mit SAVE "Programmname" abgespeichert, und lädt man dieses Programm nach Ändern des BASIC-Starts wieder in den Rechner, so wird es automatisch an die neue Adresse geladen und vom BASIC-Interpreter angepaßt, damit es probemlos auch im neuen Bereich läuft. Gibt man jedoch beim Laden die Sekundäradresse Eins an, also

```
LOAD "Programmname",1,1
```

so wird das Programm an die Adresse zurückgeladen, an der es beim Abspeichern stand, in diesem Falle also 2049. Dort ist es natürlich jetzt nicht mehr lauffähig. Das Laden an eine andere Adresse als den momentanen BASIC-Start wird dann von Bedeutung, wenn man Maschinenprogramme lädt und speichert, die ja normalerweise nicht im BASIC-Speicherbereich liegen.

Man kann auch schon beim Abspeichern festlegen, daß das Programm an die Originaladresse zurückgeladen wird, auch ohne beim LOAD-Befehl die Sekundäradresse Eins zu verwenden. Dies geschieht ganz einfach dadurch, daß man diese Sekundäradresse beim SAVE-Befehl verwendet. Beispiel: Ein mit

```
SAVE "Programmname",1,1
```

abgespeichertes Programm wird in jedem Falle an die Adresse zurückgeladen, an der es vorher im Speicher stand, egal ob man es mit

```
LOAD,
LOAD "Programmname",1
```

oder

```
LOAD "Programmname",1,1
```

einlädt. Damit läßt sich ein einfacher Schutz seiner Programme gegen Fremdbenutzung erstellen. Man verschiebt den Anfang des BASIC-Speichers an eine nur einem selbst bekannte Adresse und speichert das zu schützende Programm ab dieser Adresse unter Verwendung der Sekundäradresse Eins ab. Wer dieses Programm wieder zum Laufen bringen will, muß wissen, wo der BASIC-Start gelegen hat, da er es nicht ohne größeren Aufwand an den normalen BASIC-Start 2049 laden kann.

Zur Vollständigkeit sei hinzugefügt, daß auch beim SAVE-Befehl die Möglichkeit besteht, eine EOT-Markierung hinter das Programm zu schreiben. Dies geschieht entweder durch Angabe der Sekundäradresse 2, was dem Abspeichern ohne Sekundäradresse entspricht, oder durch Angabe der Sekundäradresse 3, entsprechend zur Sekundäradresse 1.

Hier nun nochmal eine Zusammenfassung der verschiedenen Funktionen von Sekundäradressen:

```
OPEN 1,1,0,"..."   Datenfile zum Lesen eröffnen
OPEN 1,1,1,"..."   Datenfile zum Schreiben eröffnen
OPEN 1,1,2,"..."   Datenfile zum Schreiben eröffnen mit EOT

SAVE "...",1       Programm verschiebbar abspeichern
SAVE "...",1,1     Programm unverschiebbar abspeichern
SAVE "...",1,2     Programm verschiebbar abspeichern mit EOT
SAVE "...",1,3     Programm unverschiebbar abspeichern
                   mit EOT

LOAD "...",1       verschiebbares Programm an neuen
                   BASIC-Start laden
LOAD "...",1,1     Programm an Originaladresse laden
```

5.2.2 Laden und Speichern von Maschinensprache aus

Prinzipiell läßt sich alles das, was bisher in diesem Kapitel beschrieben wurde, auch mit Maschinensprache erreichen. Durch Maschinenprogramme lassen sich aber einige wesentlich bessere Kopierschutzmethoden verwirklichen als durch BASIC-Programme. Dazu muß man wissen, wie die Entsprechungen der Befehle LOAD, SAVE, OPEN, CLOSE, GET# und PRINT# in Maschinensprache aussehen, wobei man im wesentlichen auf Unterroutinen des Betriebssystems zurückgreifen wird.

Die Befehle LOAD, SAVE und OPEN haben gemeinsam, daß sie einen Filenamen, eine Geräteadresse und eine Sekundäradresse benötigen. Intern brauchen auch LOAD und SAVE, genau wie OPEN, eine Filenummer. Um diese Parameter festzulegen, existieren zwei Betriebssystemroutinen, genannt FILPAR und FILNAM. FILPAR hat die Adresse $FFBA (=65466). Vor dem Aufruf dieser Routine lädt man die Filenummer in den Akkumulator, die Geräteadresse ins X-Register und die Sekundäradresse ins Y-Register. FILNAM beginnt bei $FFBD (=65469) und braucht im Akku die Länge des Filenamens, im X-Register das LOW-Byte der Adresse des Filenamens und im Y-Register das entsprechende HIGH-Byte. Die vollständige Sequenz, um alle Parameter zu setzen, sieht dann so aus:

Filenamenadresse: $C100
Programmstartadresse: $C000
Filename: "PROGRAMM" (muß im Speicher ab $C100 stehen)

```
C000 LDA #$01     Filenummer
C002 LDX #$01     Geräteadresse
C004 LDY #$01     Sekundäradresse
C006 JSR $FFBA    FILPAR
C009 LDA #$08     Länge des Filenamens
C00B LDX #$00     LOW-Byte von $C100
C00D LDY #$C1     HIGH-Byte von $C100
C00F JSR $FFBD    FILNAM
```

Nachdem nun das Betriebssystem alle Werte "kennt", kann man den eigentlichen Befehl aufrufen. Beginnen wir mit dem SAVE-Befehl. Die zugehörige Betriebssystemroutine beginnt bei $FFD8 (=65496). Man muß ihr allerdings den Speicherbereich mitteilen, der abgespeichert werden soll. Die Startadresse dieses Bereichs muß in zwei Zeropage-Adressen untergebracht werden, die Endadresse plus 1 im X- und Y-Register. Der Akkumulator enthält dabei die Adresse der ersten Zeropage-Speicherstelle, die man benutzt. Beispiel: Sie wollen den Speicherbereich $1000 (=4096) bis $15FF (=5631) auf Band abspeichern. Die File-Parameter sind bereits gesetzt, es fehlt nur noch das eigentliche Abspeichern. Benutzt werden sollen die Zeropage-Adressen $FB (=251) und $FC (=252). Der Aufruf könnte dann beispielsweise so erfolgen:

```
C012 LDA #$00     LOW-Byte von $1000
C014 STA $FB      nach $FB
C016 LDA #$10     HIGH-Byte von $1000
C018 STA $FC      nach $FC
C01A LDA #$FB     Benutzte Zeropage-Adresse: $FB
C01C LDX #$00     LOW-Byte von $15FF + 1
C01E LDY #$16     HIGH-Byte von $15FF + 1
C020 JSR $FFD8    SAVE
```

Das so abgespeicherte Programm läßt sich einerseits von BASIC aus mit dem Befehl LOAD wieder einladen. Man kann diesen Befehl aber auch von Maschinensprache aus benutzen. Die Adresse der Routine LOAD des Betriebssystems ist $FFD5 (=65493). Im Akku muß dabei eine Null stehen, da sonst statt der Laderoutine der Befehl VERIFY aufgerufen wird. Vorher sollte man aber wieder die Routinen FILPAR und FILNAM benutzt haben. Hat man sein Programm mit der Sekundäradresse Null abgespeichert und als Sekundäradresse für den LOAD-Befehl ebenfalls eine Null angegeben, so werden die Werte des X- und Y-Registers als LOW- und HIGH-Byte der Ladeanfangsadresse interpretiert. Beispiel: Falls beim Laden oder Speichern die Sekundäradresse Eins verwendet wurde, reichen schon folgende zwei Zeilen zum Aufruf der LOAD-Routine:

```
C012 LDA #$00     Auswahl zwischen LOAD und VERIFY
C014 JSR $FFD5    LOAD
```

Wenn dagegen beidesmal die Sekundäradresse Null gebraucht wurde, muß der Aufruf so aussehen:

Startadresse (Beispiel) : $1234

```
C012 LDA #$00     Auswahl LOAD/VERIFY
C014 LDX #$34     LOW-Byte von $1234
C016 LDY #$12     HIGH-Byte von $1234
C018 JSR $FFD5    LOAD
```

Falls während des Ladens eine Unterbrechung aufgetreten ist, so wird nach Abschluß der LOAD-Routine das Carry-Flag gesetzt. Die Unterbrechung kann beispielsweise darin bestehen, daß man während des Ladevorgangs die RUN/STOP-Taste betätigt hat, oder auch darin, daß beim Suchen nach dem nächsten Programm die Laderoutine auf eine EOT-Markierung gestoßen ist. Im letzteren Fall steht zusätzlich im Akku eine Fünf. Von der Ebene der Maschinensprache aus ist es also probemlos möglich, die EOT-Markierung abzufragen. Trotzdem sollte man diese Markierung nicht als Kopierschutz verwenden, da sie viel zu leicht zu erkennen ist.

Sollten die Daten auf dem Band fehlerhaft sein, so ist das nicht am Carry-Bit zu erkennen. Man muß in der Status-Adresse $90 (=144) nachsehen, ob sie einen Wert ungleich null enthält. Diese Adresse entspricht der BASIC-Status-Variablen ST.

Nun fehlt noch die Erklärung, wie man Datenfiles von Maschinensprache aus erzeugt. Der OPEN-Befehl benötigt, wie schon gesagt, ebenfalls einen vorhergehenden Aufruf der Routinen FILPAR und FILNAM. Danach kann man sofort die Betriebssystemroutine OPEN aufrufen, die bei $FFC0 (=65472) beginnt.

Hat man ein Datenfile zum Schreiben eröffnet, so muß man dem Computer als nächstes mitteilen, daß Bytes statt auf den Bildschirm aufs Band gebracht werden sollen. Dazu lädt man die

Filenummer ins X-Register und springt zur Routine CKOUT, $FFC9 (=65481). Jetzt lassen sich die Daten einfach dadurch auf die Kassette schreiben, daß man jeweils ein Datum in den Akku lädt und dann die Routine BASOUT bei $FFD2 (=65490) aufruft. Dies entspricht dem Schreiben von einem Zeichen mit dem Befehl PRINT# in BASIC.

Beim Einlesen des Datenfiles verfährt man entsprechend. Erst kommt wieder die Filenummer ins X-Register, worauf man das Gegenstück zu CKOUT, nämlich CHKIN, aufruft. Diese Routine beginnt bei $FFC6 (=65478). Geholt werden die Daten dann mit BASIN $FFCF (=65487). Das Datenbyte steht danach selbstverständlich im Akku. In BASIC wäre das der Befehl GET#. Achtung: Der Inhalt des Y-Registers wird beim Aufruf von BASIN zerstört.

Nach Beendigung des Lese- oder Schreibvorgangs wird der Datenkanal mit CLOSE $FFC3 (=65475) geschlossen. Die Filenummer muß dabei im Akku stehen. Am Schluß sollte man noch die Routine CLRCH $FFCC (=65484) benutzen, damit Ein- und Ausgabe wieder über Tastatur und Bildschirm erfolgen.

Nach so vielen Erklärungen nun wieder ein Beispiel: Es soll das im Speicher befindliche BASIC-Programm als Datenfile auf Band geschrieben werden. Die Anfangsadresse des Programms steht in den Speicherstellen $2B (=43) und $2C (=44), die Endadresse in $2D (=45) und $2E (=46). Als Zähler verwenden wir die Adressen $FB (=251) und $FC (=252).

```
C000 LDA #$01     Filenummer
C002 LDX #$01     Gerätenummer
C004 LDY #$01     Sekundäradresse: Schreiben auf Band
C006 JSR $FFBA    FILPAR
C009 LDA #$00     kein Filename
C00B JSR $FFBD    FILNAM
C00E JSR $FFC0    OPEN
C011 LDX #$01     Filenummer
C013 JSR $FFC9    CKOUT: Ausgabegerät Band
C016 SEC          Subtraktion vorbereiten
C017 LDA $2D      LOW-Byte der Endadresse
```

```
C019 SBC $2B       minus LOW-Byte der Anfangsadresse
C01B PHP           Carry retten
C01C JSR $FFD2     BASOUT: LOW-Byte Programmlänge
C01F PLP           Carry holen
C020 LDA $2E       HIGH-Byte der Endadresse
C022 SBC $2C       minus HIGH-Byte Anfangsadresse
C024 JSR $FFD2     BASOUT: HIGH-Byte Programmlänge
C027 LDA $2B       LOW-Byte der Anfangsadresse
C029 STA $FB       in Zähler
C02B LDA $2C       HIGH-Byte der Anfangsadresse
C02D STA $FC       in Zähler
C02F LDY #$00      indirektes Laden vorbereiten
C031 LDA $FB       schon Endadresse erreicht?
C033 CMP $2D       Test auf LOW-Byte
C035 BNE $C03D     verzweige, wenn nein
C037 LDA $FC       Endadresse erreicht?
C039 CMP $2E       Test auf HIGH-Byte
C03B BEQ $C04A     verzweige, wenn ja
C03D LDA ($FB),Y   Byte aus BASIC-Programm holen
C03F JSR $FFD2     BASOUT: Byte schreiben
C042 INC $FB       Zähler LOW-Byte erhöhen
C044 BNE $C031     verzweige, wenn kein Übertrag
C046 INC $FC       Zähler HIGH-Byte erhöhen
C048 BNE $C031     unbedingter Sprung
C04A LDA #$01      Filenummer
C04C JSR $FFC3     CLOSE
C04F JMP $FFCC     CLRCH, Abschluß
```

Das Gegenstück zu dieser Routine, welches das Programm wieder einliest, sieht so aus:

```
C000 LDA #$01      Filenummer
C002 LDX #$01      Gerätenummer
C004 LDY #$00      Sekundäradresse: Lesen vom Band
C006 JSR $FFBA     FILPAR
C009 LDA #$00      kein Filename
C00B JSR $FFBD     FILNAM
C00E JSR $FFC0     OPEN
C011 LDX #$01      Filenummer
```

```
C013 JSR $FFC6     CHKIN: Eingabegerät Band
C016 JSR $FFCF     BASIN: LOW-Byte Programmlänge
C019 STA $FB       nach Zeiger
C01B JSR $FFCF     BASIN: HIGH-Byte Programmlänge
C01E STA $FC       nach Zeiger
C020 LDA $2B       LOW-Byte Anfangsadresse
C022 STA $2D       als Startwert für Endadresse
C024 LDA $2C       HIGH-Byte Anfangsadresse
C026 STA $2E       als Startwert für Endadresse
C028 JSR $FFCF     BASIN: Zeichen holen
C02B LDY #$00      indirekte Adressierung vorbereiten
C02D STA ($2D),Y   Zeichen in Programmspeicher schreiben
C02F INC $2D       LOW-Byte Endadresse erhöhen
C031 BNE $C035     verzweige, wenn kein Übertrag
C033 INC $2E       HIGH-Byte Endadresse erhöhen
C035 DEC $FB       LOW-Byte Zähler dekrementieren
C037 LDA $FB       LOW-Byte Zähler holen
C039 CMP #$FF      Übertrag?
C03B BNE $C03F     verzweige, wenn nein
C03D DEC $FC       HIGH-Byte Zähler dekrementieren
C03F CMP #$00      LOW-Byte gleich Null?
C041 BNE $C028     verzweige, wenn nein
C043 LDA $FC       HIGH-Byte gleich Null?
C045 BNE $C028     verzweige, wenn nein
C047 LDA #$01      Filenummer
C049 JSR $FFC3     CLOSE
C04C JSR $FFCC     CLRCH
C04F JSR $A659     CLR
C052 JSR $A533     Programmzeilen binden
C055 JMP $A474     Sprung zurück ins BASIC
```

Ein auf diese Weise abgespeichertes Programm ist auf einfache Weise gegen das Kopieren gesichert. Insbesondere stehen aufgrund des Aufbaus eines BASIC-Programmtextes im Datenfile Null-Bytes, also CHR$(0)-Zeichen, wodurch es, wie schon beschrieben, schwer wird, die tatsächliche Länge des Datenfiles festzustellen.

Theoretisch könnte man nach obigem Prinzip ein Programm schreiben, das das BASIC-Programm im Speicher als Datenfile abspeichert, zusätzlich aber noch eine große Anzahl unnützer Daten. Die Laderoutine weiß, welche der Daten zum BASIC-Programm gehören und welche sie überspringen muß. Ein Kopierprogramm kann das jedoch nicht unterscheiden und muß daher zwangsläufig alles mitkopieren. Wenn man nun so viele Daten auf das Band schreibt, daß es nicht mehr möglich ist, alle diese Bytes auf einmal in den Speicher zu lesen, läßt sich das File nicht mehr kopieren. Diese Methode hat aber einen unübersehbaren Nachteil, nämlich den, daß das Datenfile so lang wird, daß es fraglich ist, ob es überhaupt noch auf eine Kassettenseite paßt. Hinzu kommt noch, daß sich die Ladezeit extrem erhöht, was sicherlich auch nicht im Sinne des Anwenders liegt.

Noch ein Letztes. Es ist von BASIC aus nicht ohne weiteres möglich (und für BASIC-Programme auch nur selten sinnvoll), die RAM-Bereiche $A000 bis $BFFF und $D000 bis $FFFF abzuspeichern. Der Bereich $D000 bis $FFFF bereitet auch von Maschinensprache aus Probleme, der Bereich $A000 bis $BFFF ist allerdings unproblematisch. Wir erwähnen das hier deshalb, weil man in die Bereiche $A000 bis $BFFF und $E000 bis $FFFF problemlos Daten hineinladen kann, was heißt, daß Programme in diesen Speicherbereichen in gewisser Weise kopiergeschützt sind. Falls Sie also ein Programm besitzen, daß länger als 38 KByte ist, also den Bereich ab $A000 belegt, so ändern Sie vor Aufruf der Routine SAVE die Speicherkonfiguration mit:

```
LDA #$36
STA $01
```

Hierdurch wird der normalerweise ab $A000 befindliche BASIC-Interpreter durch das darunter befindliche RAM ersetzt. Bevor Sie wieder wieder ins BASIC zurückkehren, ändern Sie den Wert der Speicherstelle Eins wieder auf $37 (=55):

```
LDA #$37
STA $01
```

Wie Sie auch den Bereich von $E000 bis $FFFF so abspeichern, daß er beim Einladen auch genau an diese Stelle wieder zurückgelangt, erfahren Sie im nächsten Unterkapitel.

5.3 Autostart als Kopierschutz

5.3.1 Eingriffe in die LOAD/SAVE-Routine

Für einen Kopierschutz muß man nicht unbedingt soweit gehen, daß man eine eigene Lade- und Schreibroutine entwickelt. Durch Änderungen an der bestehenden Laderoutine lassen sich auch schon effektive Schutzmechanismen erstellen. Für die folgenden Ausführungen sollten Sie allerdings einen Maschinensprachemonitor besitzen.

Es gibt prinzipiell zwei Ansätze zum Ändern der im ROM befindlichen Routinen. Man kann einerseits die LOAD- und SAVE-Vektoren, also die Speicherstellen, die die Startadressen der LOAD- und SAVE-Routinen enthalten, auf eine eigene Routine zeigen lassen. Wir wollten aber gerade keine völlig neue Routine, weshalb wir diese Methode vorerst außer acht lassen.

Der zweite Ansatz beruht darauf, daß sich im Speicherbereich des Betriebssystems umschaltbar auch noch RAM-Speicher befindet. Man kann das Betriebssystem also vom ROM ins RAM kopieren und dort dann die Änderungen vornehmen. Mit den meisten Monitoren geschieht das mit dem Befehl:

```
.T E000 FFFF E000
```

Sollte Ihr Monitor einen anderen Befehl zum Verschieben von Speicherbereichen besitzen, so verwenden Sie bitte diesen. Einige Monitore benötigen auch noch den BASIC-Interpreter im RAM, da das Betriebssystem-ROM alleine nicht abgeschaltet werden kann. In diesem Falle geben Sie auch noch folgendes ein:

```
.T A000 BFFF A000
```

Jetzt muß nur noch auf das RAM umgeschaltet werden. Geben Sie zuerst ein:

```
.M 0001
```

Es sollten jetzt acht Hexadezimalzahlen angezeigt werden, von denen die erste den Wert $37 hat. Ändern Sie bitte diesen Wert auf $35 und vergessen Sie bitte nicht, RETURN zu drücken. Jetzt kann das Betriebssystem nach Belieben abgeändert werden.

Als erstes wollen wir das Problem lösen, Programme so abzuspeichern, daß sie beim Einladen nach $E000 gelangen. Bringen Sie dazu das Programm in einen freien Teil des Speichers, beispielsweise $1000. Angenommen dieses Programm sei $1234 Bytes lang, dann liegt es normalerweise im Bereich $E000 bis $F234. Modifizieren Sie nun mit dem Assembler Ihres Monitors die Abspeicherroutine wie folgt:

```
.A F78B LDA #$00      LOW-Byte von $E000
.A F790 LDA #$E0      HIGH-Byte von $E000
.A F795 LDA #$35      LOW-Byte von $F234 + 1 = $F235
.A F79A LDA #$F2      HIGH-Byte von $F235
```

Wenn Sie jetzt das Programm von $1000 bis $2234 abspeichern, kommt es beim Einladen nach $E000. Bevor Sie weiterarbeiten, sollten Sie aber die SAVE-Routine wieder restaurieren, indem Sie das ROM wieder einschalten, also $37 in die Speicherstelle 1 schreiben. Falls Sie direkt im Anschluß daran andere Veränderungen am Betriebssystem vornehmen wollen, so kopieren Sie besser erst wieder das ROM ins RAM.

Theoretisch könnte man durch Modifikationen am Betriebssystem das Aufzeichnungsformat völlig ändern oder die Aufzeichnungsgeschwindigkeit erhöhen. Die Lade- und Speicher-Routinen des Systems sind aber dermaßen kompliziert aufgebaut, daß es wesentlich (!) einfacher ist, für solche Zwecke eine komplett neue Routine zu entwerfen.

Wir wollen hier nicht weiter auf das weite Feld der Betriebssystemänderungen eingehen, da wir uns sonst vom Hauptthema

diese Kapitels, dem Kopierschutz, zu weit entfernen. Falls Sie selbst eigene Verbesserungen am Betriebssystem vornehmen wollen, empfehlen wir Ihnen, sich ein dokumentiertes ROM-Listing zu besorgen, wie es beispielsweise im '64 Intern' abgedruckt ist.

5.3.2 Selbststartende Programme

Nun aber zu einer echten Kopier-/Programmschutzanwendung, dem Autostart. Im Gegensatz zur Diskettenstation stellt ein einfacher Autostart bei der Datasette schon einen Kopierschutz dar, da es kaum ein Kopierprogramm gibt, das ein so ausgestattetes Programm kopieren kann.

Es gibt noch einen anderen Unterschied zum Diskettenlaufwerk: Die Möglichkeit für einen Autostart sind nicht so vielfältig. Ein Autostart wird grundsätzlich folgendermaßen erzeugt: Man ändert zuerst den Vektor einer Betriebssystem- oder BASIC-Interpreter-Routine, die sofort nach dem Ladevorgang aufgerufen wird, so ab, daß er auf das eigene Programm zeigt. Dann speichert man das Programm mit dem veränderten Vektor ab. Dabei ist es wichtig, die Sekundäradresse Eins zu verwenden, damit beim Einladen die Daten nicht an den BASIC-Anfang, also nach $0801, kommen. Am Schluß sollte man die Vektoren wieder zurücksetzen. Nach dem Einladen eines so aufgenommenen Programms springt der Prozessor über den umgestellten Vektor ohne weiteres Dazutun in das Programm, welches als erstes den benutzten Zeiger auf seinen ursprünglichen Wert setzen sollte. Wer will, kann den Vektor für die Abfrage der STOP-Taste zusammen mit dem Einsprungvektor umändern und abspeichern, wodurch das Programm gegen Eingriffe durch STOP oder STOP/RESTORE gesichert wird.

Wo liegen die benötigten Vektoren im Speicher? Für einen Autostart kommen nur drei Zeiger in Frage:

Vektor	norm. Wert	Funktion
$0302/$0303	$A483	Eingabe einer BASIC-Zeile
$0324/$0325	$F157	BASIN, Eingabe eines Zeichens
$0326/$0327	$F1CA	BASOUT, Ausgabe eines Zeichens

Außerdem ist noch der STOP-Vektor von Bedeutung:

Vektor	norm. Wert	Wert für gesperrte STOP-Taste
$0328/$0329	$F6ED	$F6E1

Der Vektor $0326/$0327 ist eigentlich für einen Autostart nach der oben beschriebenen Methode nicht zu gebrauchen, da das Programm schon anläuft, sobald nur die Meldung 'PRESS RECORD AND PLAY ON TAPE' ausgegeben werden soll. Wir werden aber trotzdem noch auf ihn zurückkommen.

Wenn man auf die oben beschriebene Weise einen Autostart erzeugt, muß man beachten, daß man nicht den Vektor $0314/$0315 (IRQ-Vektor) mitabspeichert, da dieser Vektor sowohl von der LOAD- als auch von der SAVE-Routine verstellt wird. Das bedeutet, daß man für Programme im Bereich $02A7 bis $02FF den Vektor $0302/$0303 benutzen muß, dagegen für Programme ab $0801, insbesondere BASIC-Programme, auf den Vektor $0324/$0325 zurückgreifen muß. Es läßt sich dabei nicht umgehen, daß der von $0400 bis $07E7 befindliche Bildschirmspeicher mitabgespeichert wird. Wie schon erwähnt, steht die Endadresse plus 1 des aktuellen BASIC-Programms in den Speicherstellen $2D (=45) und $2E (=46). Außerdem existieren zwei Betriebssystemroutinen zum Zurücksetzen der benutzten Vektoren:

```
$E453 initialisiert Vektoren $0300 bis $030B
$FF8A initialisiert Vektoren $0314 bis $0333
```

Das Abspeichern sieht dann prinzipiell so aus:

```
LDA #$Startadresse LOW-Byte
STA $0324          BASIN-Vektor LOW
LDA #$Startadresse HIGH-Byte
STA $0325          BASIN-Vektor HIGH
LDA #$E1           STOP-Vektor: $F6ED in $F6E1 umändern
STA $0328          und dadurch STOP sperren.
LDA #$01           Filenummer 1
TAX                Geräteadresse 1
TAY                Sekundäradresse 1
JSR $FFBA          FILPAR
LDA #$Filenamenlänge
LDX #$Filenamenadresse LOW-Byte
LDY #$Filenamenadresse HIGH-Byte
JSR $FFBD          FILNAM
LDA #$26           LOW-Byte $0326
STA Zeiger auf Anfangsadresse
LDA #$03           HIGH-Byte $0326
STA Zeiger auf Anfangsadresse + 1
LDA #$Zeiger auf Anfangsadresse
LDX $2D            Endadresse LOW-Byte
LDY $2E            Endadresse HIGH-Byte
JSR $FFD8          SAVE
JSR $FF8A          Vektoren initialisieren
RTS
```

Das so abgespeicherte Programm sollte nicht die Routine $FF8A zum Zurücksetzen des Autostart-Vektors nehmen, da damit auch der STOP-Vektor wieder auf seinen ursprünglichen Wert zurückgesetzt würde. Mit dieser Methode kann man allerdings nur Maschinenprogramme automatisch starten. Wollen Sie jedoch ein BASIC-Programm selbsttätig anlaufen lassen, so müssen Sie eine Maschinenroutine mitabspeichern, die den RUN-Befehl des BASIC-Interpreters aufruft. Dieses Programm sollte am besten im freien Bereich $07E8 bis $07FF liegen. Die Sequenz für RUN sieht so aus:

```
LDA #$00
JSR $A871
JMP $A7AE
```

Es gibt noch eine andere Möglichkeit, das Abspeichern zu organisieren. Man kann sein Programm inklusive den (unveränderten) Vektoren in einen freien Speicherbereich verschieben, am besten von $0324 beginnend nach $1324. Dann verändert man das Betriebssystem in der oben beschriebenen Weise so ab, daß ein abgespeichertes Programm in jedem Fall nach $0324 geladen wird. Man kann nun ohne weiteres mit einem Monitor die Vektoren ab $1324 nach Belieben umstellen, beispielsweise jetzt auch den Vektor $0326/$0327. Es ist auch möglich, Programme in den Bereich $133C bis $17E7 einzubringen, die nach dem Einladen in $033C bis $07E7 stehen, also im Bandpuffer oder im Bildschirmspeicher. Andererseits besteht kein Zwang, bei $0324 zu beginnen. Genausogut kann man durchgehend den Bereich von $02A7 bis zum Programmende nach $12A7 bringen und von dort aus abspeichern. Dabei muß allerdings der IRQ-Vektor bei $1314/$1315 auf den Wert $F92C gesetzt werden.

Ein Nachteil dieser Schutzmethoden soll hier nicht verschwiegen werden. Mit den gleichen Tricks, mit denen man einen Autostart erzeugt, kann er auch wieder entfernt werden. Es ist für einen Knacker sicherlich ein leichtes, eine Routine zu schreiben, die ein selbsstartendes Programm einlädt und danach sofort alle Vektoren initialisiert. Insofern ist es auch sicherer, die Vektoren $0324/$0325 und $0326/$0327 dem Vektor $0302/$0303 vorzuziehen, da letzterer nicht funktioniert, wenn das geschützte Programm mit einem Monitor geladen wird.

Zum Schluß noch ein Programm, das ein beliebiges mit RUN startbares Programm mit einem Autostart versieht. Das Programm benutzt die BASIC-Interpreterroutine $E1D4, welche die Parameter für LOAD und SAVE, also Filename, Geräteadresse und Sekundäradresse, aus der eingegebenen BASIC-Zeile holt. Dadurch läßt es sich folgendermaßen bedienen: Tippen Sie zuerst den abgedruckten BASIC-Lader ein und starten Sie ihn

mit RUN. Danach laden Sie das zu schützende Programm. Geben Sie dann bitte 'SYS 49152 "Filename"' ein. Ihr Programm wird nun zusammen mit einem Autostart auf Band gebracht.

BASIC-Lader:

```
100 FORI=1TO85STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATAA2,12,BD,43,C0,9D,E8,07,CA,10,F7,20,D4,E1,A9, 79
301 DATA01,AA,A8,20,BA,FF,A9,93,20,D2,FF,A9,E8,8D,24, 155
302 DATA03,A9,07,8D,25,03,A9,E1,8D,28,03,A9,36,85,01, 15
303 DATAA9,24,85,FB,A9,03,85,FC,A9,FB,A6,2D,A4,2E,20, 227
304 DATAD8,FF,E6,01,4C,8A,FF,A9,57,8D,24,03,A9,F1,8D, 110
305 DATA25,03,A9,00,20,71,A8,4C,AE,A7,A4,A4,A4,CC,C4, 39
```

Assemblerlisting:

```
benutzte Adressen:
$FB/$FC: Zeiger auf Startadresse für SAVE

C000 LDX #$12       Zähler für Schleife auf $12 setzen
C002 LDA $C043,X    Start-Programm, welches RUN- Befehl
C005 STA $07E8,X    ausführt, nach $07E8 verschieben
C008 DEX            schon alle Bytes verschoben?
C009 BPL $C002      verzweige, wenn nein
C00B JSR $E1D4      Filenamen holen
C00E LDA #$01       Filenummer = 1
C010 TAX            Geräteadresse = 1
C011 TAY            Sekundäradresse = 1
C012 JSR $FFBA      FILPAR
C015 LDA #$93       $93 ist ASCII-Wert für 'CLR/HOME'
C017 JSR $FFD2      BASOUT, hier: Bildschirm löschen
C01A LDA #$E8       $07E8
C01C STA $0324      in den
```

```
C01F LDA #$07       BASIN-Vektor
C021 STA $0325      schreiben
C024 LDA #$E1       STOP-Vektor
C026 STA $0328      auf $F6E1 setzen
C029 LDA #$36       im Bereich $A000 bis $BFFF
C02B STA $01        das RAM einschalten
C02D LDA #$24       $0324 als
C02F STA $FB        Startadresse
C031 LDA #$03       für SAVE
C033 STA $FC        nach $FB/$FC
C035 LDA #$FB       $FB ist der benutzte Zeiger
C037 LDX $2D        Programmendadresse LOW-Byte
C039 LDY $2E        Programmendadresse HIGH-Byte
C03B JSR $FFD8      SAVE
C03E INC $01        ROM wieder einschalten ($37 nach $01)
C040 JMP $FF8A      Vektoren initialisieren
```

Die folgende Routine wird nach $07E8 verschoben, mitabgespeichert und vom Autostart angesprungen.

```
C043 LDA #$57       BASIN-Vektor auf
C045 STA $0324      seinen alten
C048 LDA #$F1       Wert $F157
C04A STA $0325      zurücksetzen
C04D LDA #$00       Zero-Flag setzen
C04F JSR $A871      RUN-Befehl aufrufen
C052 JMP $A7AE      zurück zum BASIC-Interpreter
```

5.4 Der Kassettenpuffer

Der Kassetten- oder Bandpuffer ist ein Speicherbereich, der zwei Bestimmungen hat. Zum einen enthält er beim Laden und Abspeichern den schon angesprochenen "Header" (engl. Head = Kopf), also die Daten über Start- und Endadresse und den Filenamen des Programms. Zum anderen wird er bei Datenfiles dazu benutzt, den jeweils nächsten Datenblock aufzunehmen. Er

liegt normalerweise im Bereich $033C (=828) bis $03FF (=1023). In seiner Funktion als Header-Speicher ist seine Aufteilung folgende:

Hexadez.	Dezimal	Funktion
033C	828	Headertyp
033D/033E	829/ 830	Startadresse des Programmes
033F/0340	831/ 832	Endadresse des Programmes
0341-03FF	833-1023	Filename

Der Puffer muß nicht bei 828 beginnen, er kann auch verschoben werden. Der Zeiger $B2/$B3 (178/179) weist auf seinen Beginn. Durch Ändern dieses Zeigers läßt sich aber kein wirkungsvoller Kopierschutz aufbauen, weshalb wir darauf auch nicht eingehen werden.

Als erstes Byte des Kassettenpuffers ist der Header-Typ vermerkt. Folgende Werte haben dabei die folgende Bedeutung:

1: verschiebbar abgespeichertes Programm; bei Verwendung der Sekundäradresse 0 wird das Programm nicht an die im Header angegebene Startadresse geladen, sondern an die vom Anwender übergebene. Beim BASIC-Befehl LOAD ist das der Anfang des BASIC-Speichers $0801 (=2049).

2: dieser Block ist kein Header, sondern ein Datenblock. Er enthält keine Adressen und keinen Filenamen, sondern nur Datenbytes.

3: nicht verschiebbar abgespeichertes Programm; das Programm wird in jedem Fall an die im Header angegebene Adresse geladen.

4: Header eines Datenfiles

5: Header einer End-of-Tape-Markierung.

Die nächsten vier Bytes beinhalten die Start- und Endadressen. Probieren Sie doch beispielsweise folgendes aus: Laden Sie ein

auf einer Kassette abgespeichertes Programm bis zur Anzeige der 'FOUND'-Meldung und betätigen Sie dann die STOP-Taste. Durch 'PEEK(829) + PEEK(830) * 256' erhalten Sie dann die Startadresse, durch 'PEEK(831) + PEEK(832) * 256' die Endadresse des Programms.

Alle weiteren Bytes sind für den Filenamen zuständig. Sie werden sich vielleicht darüber wundern, daß hier 191 Bytes frei sind, obwohl der Filename ja normalerweise nur 16 Zeichen lang sein darf. In der Tat werden beim Laden auch nur maximal 16 Zeichen als Name angezeigt. In Wirklichkeit kann der Filename jedoch wesentlich länger sein. Geben Sie zum Beispiel folgendes ein:

```
SAVE "LANGER NAME      Z"
```

Achten Sie bitte darauf, daß das einzelne "Z" den 17. Buchstaben bildet. Wenn Sie das so abgespeicherte Programm wieder einladen, erscheint als Meldung:

```
FOUND LANGER NAME
```

Das "Z" ist also nicht mehr zu sehen. Trotzdem erhalten Sie mit 'PRINT CHR$(PEEK(849))' ein "Z" zurück, das heißt, der Buchstabe steht an der richtigen Stelle im Bandpuffer. Dieser Trick stellt schon einen eleganten Kopierschutz dar, sofern man in seinem Programm einen Test auf die Korrektheit des Filenamens vornimmt. Schließlich kann ein Kopierer ohne genauere Überprüfung nicht feststellen, welchen Namen er dem Programm geben muß, damit es läuft.

Man kann das Spiel sogar noch weiter treiben, indem man auf diese Weise ganze Programme im Kassettenpuffer unterbringt. Es ist sogar möglich, dort eine komplette eigene Laderoutine unterzubringen, wodurch der gesamte restliche Speicher für das eigentliche Programm frei bleibt. Dazu gehen Sie folgendermaßen vor: Schreiben Sie zuerst das Programm für den Puffer beginnend bei $0351 (=849). Hier fängt nämlich der nicht sichtbare Teil des Filenamens an. Schieben Sie dann dieses Programm direkt hinter den 16 Zeichen langen Filenamen, den Sie vorher

in einem freien Teil des Speichers abgelegt haben. Sollte der Filename weniger als 16 Zeichen besitzen, so füllen Sie die fehlenden Bytes mit Leerzeichen auf. Beim Abspeichern rufen Sie wie gewohnt die Routinen FILPAR, FILNAM und SAVE auf (s. Unterkapitel 2.2). Die Filenamenlänge geben Sie dabei groß genug an, damit das gesamte Programm erfaßt wird. Damit bei der Meldung 'SAVING ...', die sich im Gegensatz zur FOUND-Meldung nicht auf 16 Buchstaben beschränkt, keine unnötigen Zeichen auf dem Bildschirm erscheinen, können Sie vorher noch die Speicherstelle $9D (=157) auf Null setzen. Damit verhindern Sie die Ausgabe der Betriebssystemmeldungen. Das im Unterkapitel 5.3 beschriebene Kassetten-Kopierschutz-System liefert ein Beispiel für dieses Vorgehen.

In Verbindung mit der Erstellung von Datenfiles war davon die Rede, daß man Daten ohne File-Endemarkierung schreiben kann. Dazu muß man wissen, daß für diese Form der Datenspeicherung der Bandpuffer von $033C (=828) bis $03FB (=1019) benutzt wird. Wenn man nun nur einige Daten mit dem Befehl 'PRINT#' in das File und damit an den Anfang des Puffers schreibt, wird bei Aufruf des 'CLOSE'-Befehls trotzdem immer der gesamte Pufferinhalt aufs Band geschrieben. Daher lassen sich Werte in den Kassettenpuffer "poken", die zwar nicht mit einer Null abgeschlossen sind, aber trotzdem abgespeichert werden. Das ganze sieht dann so aus:

```
OPEN 1,1,1
PRINT#1,"DATEN"
POKE 1010,123
CLOSE 1
```

Nach der 123 steht keine Null als File-Endekennzeichen. Das Einlesen geschieht analog dazu:

```
OPEN 1,1,0
A=PEEK(1010)
```

A enthält jetzt den Wert 123, sofern niemand den gescheiterten Versuch unternommen hat, das Datenfile zu kopieren.

5.5 Entwicklung eines eigenen Aufzeichnungsformats

5.5.1 Direkte Ansteuerung der Datasetten-Funktionen

Die wohl interessanteste und sicherste Schutzmethode ist die Entwicklung einer eigenen Aufzeichnungs- und Laderoutine. Neben der Tatsache, daß dieses Verfahren Kopierversuche ohne Zuhilfenahme zweier Rekorder völlig sinnlos macht, spricht noch ein anderer Grund für diesen aufwendigen Schutz. Er erlaubt nämlich, die Ladegeschwindigkeit beträchtlich zu erhöhen, was sich bei längeren Programmen äußerst angenehm bemerkbar macht. Weiterhin bietet eine eigene Laderoutine meist wesentlich mehr Möglichkeiten für einen guten Programmschutz als die Originalroutine. Es ist zum Beispiel möglich, Programmteile in die gerade ablaufende Laderoutine hineinzuladen, ohne die man die nachfolgenden Daten nicht lesen kann. Man kann damit ein solches Verwirrspiel treiben, daß auch hartnäckigste Knacker nach einiger Zeit den Mut verlieren. Wir kommen allerdings nicht darum herum, einige Grundlagen zu beschreiben, damit Sie die weiteren Erklärungen verstehen.

Wie sieht prinzipiell das Schreiben und Lesen von Daten auf Band aus? Man braucht eigentlich nur eine Technik, die es einem erlaubt, einzelne Bits auf dem Band abzulegen. Schließlich kann man sein Programm komplett in Bits 'zerlegen', die dann beim Einladen wieder zu Programm-Bytes zusammengefügt werden. Tatsächlich existiert auch eine Leitung, über die ein LOW- oder HIGH-Signal aufs Band geschrieben werden kann. Man kann aber jetzt nicht hingehen und einfach jedes Bit, das man schreiben will, auf diese Leitung legen. Wenn man nämlich mehrere gleichartige Bits hintereinander schreiben würde, ließe sich aufgrund der relativ hohen Gleichlaufschwankungen der Datasette nicht mehr exakt unterscheiden, wieviele Bits sich auf dem Band befinden. Außerdem existiert keine Leitung, an der man direkt das LOW- oder HIGH-Signal auf dem Band ablesen könnte. Ein lesbares Signal stellt nur der Wechsel von HIGH nach LOW dar, was als fallende beziehungsweise negative Flanke bezeichnet wird. Um nun lesbare Daten auf eine Kassette zu

bringen, verfährt man wie folgt: Man untersucht fallende Flanken nach unterschiedlichen Zeitabständen, wobei beispielsweise eine kleine Zeitspanne ein Null-Bit und eine große Zeitspanne ein Eins-Bit bedeutet. Beim Einlesen mißt man diese Zeitspanne und kann dann daraus erkennen, ob ein Null- oder Eins-Bit auf dem Band steht.

Timing beim Lesen und Schreiben von Bits auf Band

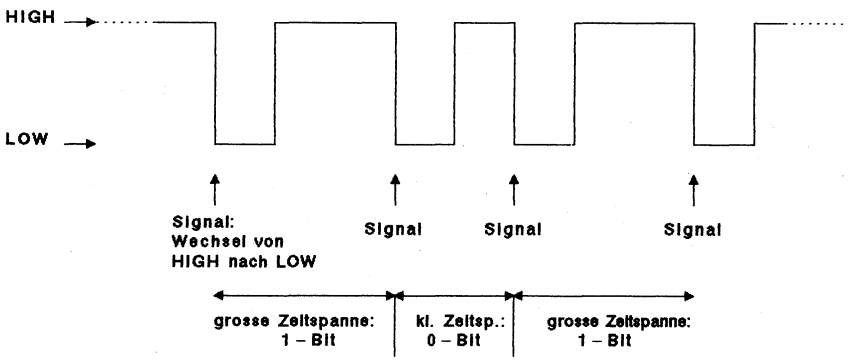


Abb. 5.5.1: Timing beim Lesen und Schreiben von Bits auf Band

Da sich zum Messen von Zeiten ein Chip im C64 besonders eignet, der sogar gleich zweimal vorhanden ist, soll dieser Chip als allererstes beschrieben werden.

Die Rede ist vom 'Complex Interface Adapter 6526', kurz CIA. Für unsere Zwecke reicht es, nur den ersten CIA zu betrachten. Die Basisadresse dieses Chips liegt bei $DC00 (=56320). Die für uns interessanten Register sind die folgenden:

```
$DC04 (=56324): Timer A LOW-Byte
$DC05 (=56325): Timer A HIGH-Byte
$DC06 (=56326): Timer B LOW-Byte
$DC07 (=56327): Timer B HIGH-Byte
```

```
$DC0D (=56333): Interrupt-Kontroll-Register (ICR)
$DC0E (=56334): Kontroll-Register Timer A (CRA)
$DC0F (=56335): Kontroll-Register Timer B (CRB)
```

Der CIA enthält zwei 16-Bit-Timer, die von einem voreingestellten Wert auf Null zurückzählen und dann einen Interrupt auslösen können. Timer A wird vom Betriebssystem verwendet, um in regelmäßigen Abständen die Tastatur abzufragen. Man sollte daher für eigene Experimente Timer B verwenden.

Jeder der beiden Timer besteht aus einem Speicher für den Startwert und einem Zähler. Schreibt man in eines der Timer-Register einen Wert, so gelangt dieser in den Startwertspeicher. Liest man das Timer-Register aus, so erhält man den aktuellen Stand des Zählers. Ob der Zähler gestartet, angehalten oder auf den Startwert gesetzt werden soll, bestimmt man über das zugehörige Kontrollregister. Ebenfalls über das Kontrollregister wird die Timer-Triggerung festgelegt, das heißt, das Signal, welches das Herabzählen um Eins auslöst. Üblicherweise geschieht dies durch den Sytemtakt oder auch durch externe Signalquellen. Es besteht aber auch die Möglichkeit, Timer B zählen zu lassen, wie oft Timer A abgelaufen ist. Dadurch kann man die zwei 16-Bit-Timer zu einem 32-Bit-Timer zusammenschalten. Hier nun die genaue Beschreibung der Kontrollregister:

```
Kontrollregister A: $DC0E
Bit 0:   1=Timer A Start, 0=Timer A Stop
Bit 1-2: nur für externe Signale wichtig
Bit 3:   1=Timer zählt nur einmal vom Startwert auf Null,
         dann wird der Startwert wieder in den Zähler
         geladen und der Timer gestoppt.
         0=Timer zählt fortlaufend vom Startwert auf Null
         und beginnt dann wieder von vorn.
Bit 4:   1=Startwert wird in Zähler übertragen, egal, ob
         der Timer gerade läuft oder nicht.
Bit 5:   0=Timer zählt Systemtaktpulse.
         1=Timer wird durch eine externe Signalquelle
         getriggert.
Bit 6-7: für unser Vorhaben uninteressant.
```

```
Kontrollregister B: $DC0F
Bit 0-4: Diese Bits haben die gleiche Bedeutung wie die
         Bits 0 bis 4 von Timer A, nur jetzt bezogen auf
         Timer B.
Bit 5-6: 00=Timer zählt Systemtaktpulse.
         01=Timer wird durch eine externe Signalquelle
         getriggert.
         10=Timer B zählt, wie oft Timer A abgelaufen ist.
         11=Timer B zählt, wie oft Timer A abgelaufen ist,
         aber nur, wenn ein externes Signal anliegt.
Bit 7:   für unser Vorhaben uninteressant.
```

Über das Interrupt-Kontroll-Register läßt sich festlegen, ob ein bestimmtes Ereignis, wie zum Beispiel das Ablaufen eines Timers, zu einem Interrupt (IRQ) am Prozessor führen soll. Außerdem kann man über dieses Register feststellen, ob eines dieser Ereignisse schon stattgefunden hat. Hier nun die Zuordnung der Bits dieses Registers zu den möglichen Ereignissen:

```
Interrupt-Kontroll-Register: $DC0D
Bit 0:   1=Timer A abgelaufen.
Bit 1:   1=Timer B abgelaufen.
Bit 2-3: für unser Vorhaben uninteressant.
Bit 4:   1=negative Flanke am Pin-FLAG aufgetreten. Da der
         Pin-FLAG dieses Chips mit der Ausgangsleitung des
         Kassettenrekorders verbunden ist, läßt sich anhand
         dieses Bits feststellen, ob ein Signal auf dem
         Band aufgezeichnet wurde.
Bit 5-6: immer Null.
Bit 7:   1=Interrupt wurde durch eines der oben
         aufgeführten Ereignisse ausgelöst.
```

Wenn man in dieses Register einen Wert schreibt, beeinflußt man den sogenannten Interrupt-Masken-Speicher, welcher bestimmt, welche(s) Ereignis(se) zu einem IRQ führen soll(en).

Wenn Bit 7 des geschriebenen Wertes gesetzt ist, werden alle Bits des Masken-Speichers gesetzt, die in diesem Wert auf Eins standen. Entsprechend kann man die Bits des Masken-Speichers löschen, indem man Bit 7 auf Null setzt und alle zu löschenden Bits auf Eins. Dazu ein Beispiel:

```
LDA #$81
STA $DC0D
```

setzt Bit 0 im Maskenspeicher, da $81=%10000001 ist. Nur Bit 0 wird gesetzt, alle anderen Bits bleiben unbeeinflußt.

```
LDA #$02
STA $DC0D
```

löscht Bit 1, da $02=%00000010 ist. Ein gesetztes Bit im Masken-Speicher läßt das zugehörige Ereignis einen Interrupt auslösen. Ein gesetztes Bit im Interrupt-Kontroll-Register dagegen zeigt an, ob ein Ereignis stattgefunden hat. Diese Bits werden automatisch gelöscht, sobald man dieses Register ausliest. Es ist allerdings auch notwendig, die Bits zu löschen, wenn ein Interrupt durch das entsprechende Ereignis stattfinden soll. Das bedeutet insbesondere, daß eine regelmäßig aufgerufene Interrupt-Routine jedesmal durch einen Lesezugriff die Bits auf Null setzen muß.

Theoretisch könnten wir jetzt schon Daten vom Band lesen. Was uns aber dazu noch fehlt, ist die Methode, wie wir überhaupt Daten aufs Band bekommen. Wie gesagt, existiert eine Leitung, deren LOW- oder HIGH-Signal direkt zur Datasette geschickt wird. Diese Leitung ist mit dem Ein-/Ausgabe-Port des Prozessors verbunden, das heißt, sie läßt sich über die Speicherstelle Eins ansprechen. Über diese Speicherstelle läßt sich auch der Motor der Datasette einschalten und testen, ob eine Taste des Recorders gedrückt ist. Hier nun die Funktionen der einzelnen Bits:

```
Prozessor-Port: $01
Bit 3: Schreib-Leitung zur Datasette
Bit 4: 0=Datasetten-Taste gedrückt, 1=nicht gedrückt
Bit 5: 0=Motor ein, 1=Motor aus
```

Noch ein Hinweis, bevor wir zu Beispielen übergehen: Wie bei allen zeitkritischen Operationen muß man vor Beginn unserer Schreib- und Lesevorgänge dafür sorgen, daß der Prozessor nicht von außen unterbrochen oder angehalten wird. Falls Sie also keine eigene Interrupt-Routine für den Bandbetrieb verwenden, benutzen Sie am Beginn Ihres Programms den Befehl SEI. Außerdem stoppt der Videocontroller (VIC) bei eingeschaltetem Bildschirm den Prozessor von Zeit zu Zeit für 42 Takzyklen. Um den Bildschirm abzuschalten, löscht man Bit 4 im Register $D011 (=53265) beispielsweise so:

```
LDA $D011
AND#$EF
STA $D011
```

Das Wiedereinschalten des Bildschirms sieht dann entsprechend so aus:

```
LDA $D011
ORA#$10
STA $D011
```

5.5.2 Signalerzeugung auf dem Band als Kopierschutz

Jetzt wollen wir endlich dazu übergehen, das theoretisch Besprochene in die Praxis umzusetzen. Wir werden ein Programm entwerfen, das eine Reihe von Signalen in gleichen Zeitabständen aufs Band schreibt. Das dazu passende Leseprogramm soll erkennen, ob die Abstände zwischen den Signalen stimmen oder nicht. Diese Aufzeichnung kann demnach schon als Kopierschutz fungieren, indem man sie einfach hinter sein abgespeichertes Programm setzt und abfragt. Das eigentliche Programm läßt sich dann zwar kopieren, jedoch der 'von Hand' erzeugte Teil nicht. Hier also das Programm:

benutzte Speicherstellen:
$FB/$FC : Zähler für die Anzahl geschriebener Signale

```
C000 SEI          Interrupts verhindern
C001 LDA $D011    Unterbrechungen durch
C004 AND #$EF     den Videocontroller
C006 STA $D011    verhindern
C009 LDA #$10     Bit 4 von Speicherstelle 1 testen
C00B BIT $01      Datasetten-Taste gedrückt?
C00D BNE $C00B    verzweige, wenn nein
C00F LDA $01      Speicherstelle 1 auslesen
C011 AND #$DF     Bit 5 löschen (Motor an)
C013 STA $01      Wert zurückschreiben
C015 LDA #$04     Zähler für Zeitschleife
C017 LDX #$00     Zähler für Zeitschleife
C019 DEY          abwarten, bis der Motor
C01A BNE $C019    seine Arbeitsgeschwindigkeit
C01C DEX          erreicht hat
C01D BNE $C019    dabei gleichzeitig Abstand
C01F SEC          für die Leseroutine halten
C020 SBC #$01
C022 BNE $C019
C024 STX $FB      Zähler $FB/$FC
C026 LDA #$20     auf $2000 (=8192)
C028 STA $FC      setzen
C02A LDA $01      Bit 3 von Speicherstelle 1
C02C AND #$F7     löschen
C02E STA $01      (LOW-Signal auf dem Band erzeugen)
C030 LDX #$14     Zeitschleifenzähler auf $14 (=20)
C032 DEX          20*5=100 Taktzyklen warten
C033 BNE $C032
C035 ORA #$08     Bit 3 von Speicherstelle 1 setzen
C037 STA $01      (HIGH-Signal auf dem Band erzeugen)
C039 LDX #$3C     Zeitschleifenzähler auf $3C (=60)
C03B DEX          60*5=300 Taktzyklen warten
C03C BNE $C03B
C03E DEC $FB      Zähler LOW-Byte verringern
C040 BNE $C02A    verzweige, wenn noch nicht Null
```

```
C042 DEC $FC        Zähler HIGH-Byte verringern
C044 BNE $C02A      verzweige, wenn noch nicht Null
C046 LDA $D011      Bildschirm durch Setzen
C049 ORA #$10       von Bit 4 in $D011
C04B STA $D011      wieder einschalten
C04E LDA $01        Motor der Datasette
C050 ORA #$20       durch Setzen von Bit 5 in Adresse 1
C052 STA $01        abschalten
C054 STA $C0        Motorsteuerflag ungleich Null
C056 CLI            Interrupts wieder zulassen
C057 RTS            Rücksprung
```

Zuerst sperrt dieses Programm jegliche Unterbrechungsmöglichkeiten. Dann fragt es Bit 4 von Speicherstelle Eins ab, um zu sehen, ob schon eine Taste an der Datasette gedrückt wurde. Sobald man RECORD und PLAY betätigt hat, wird durch Löschen von Bit 5 in Adresse Eins der Datasettenmotor eingeschaltet. Man muß dem Motor etwas Zeit geben, bis er seine volle Geschwindigkeit erreicht. Dazu reicht normalerweise die oben verwendete Zeitschleife ab $C017 aus:

```
C017 LDX #$00
C019 DEY
C01A BNE $C019
C01C DEX
C01D BNE $C019
```

Hier geben wir aber dem Motor sogar die vierfache Zeit, damit später beim Lesen der Daten ein größerer Spielraum für die Hochlaufzeit vorhanden ist und der Anfang der Signale nicht verpaßt wird.

Als nächstes setzt das Programm den Zähler für die Anzahl der zu schreibenden Signale auf $2000 (=8192). Dies scheint ziemlich groß, aber da der zeitliche Abstand zwischen zwei Signalen etwa 400 Taktzyklen beträgt, benötigt die Aufzeichnung nur knapp dreieinhalb Sekunden. Um ein Signal zu schreiben, wird für etwa 100 Taktzyklen die Schreibleitung der Datasette auf LOW gelegt. Danach erhält diese Leitung für ungefähr 300 Taktzyklen

den Pegel HIGH. Der Abstand zwischen den Signalen beträgt demnach 400 Zyklen. Man sollte einen LOW- oder HIGH-Pegel für mindestens 30 bis 40 Taktzyklen beibehalten, da aufgrund der Gleichlaufschwankungen des Recorders sonst die Gefahr besteht, daß ein Signal beim Lesen nicht mehr erkannt wird. Um die richtige Anzahl von Taktzyklen zwischen den Signalen abzuwarten, verwenden wir folgende Form der Zeitschleife:

```
        LDX #$Wert
LOOP    DEX
        BNE LOOP
```

Wie Sie aus der Taktzyklentabelle dieses Buches entnehmen können, benötigt der Befehl 'DEX' zwei, der Befehl 'BNE' (wenn die Verzweigung ausgeführt wird) drei Zyklen. Die obige Befehlsfolge benötigt demnach ziemlich genau 'Wert * 5' Taktzyklen (eigentlich: Wert * 5 + 1).

Nachdem alle Signale geschrieben wurden, wird der Motor der Datasette abgeschaltet, der Bildschirm wieder eingeschaltet und der Wert der Speicherstelle $C0 (=192) ungleich Null gesetzt. Diese Speicherstelle wird vom Betriebssystem dazu verwendet, um zu bestimmen, ob bei gedrückter Rekorder-Taste der Motor gestartet werden soll oder nicht. Falls man diese Speicherstelle auf dem Wert Null ließe, so würde nach dem Befehl 'CLI' die Datasette, trotz vorherigem Setzen des Bits 5 der Speicherstelle Eins, weiterlaufen.

Zu diesem Programm benötigt man ein zweites, welches die Daten wiedererkennt:

benutzte Speicherstellen: siehe oben

```
C000 SEI          Interrupts verhindern
C001 LDA $D011    Unterbrechungen durch
C004 AND #$EF     den Videocontroller
C006 STA $D011    verhindern
C009 LDA #$FF     Startwert für Timer $FFFF
C00B STA $DC06    Timer LOW-Byte setzen
```

```
C00E STA $DC07      Timer HIGH-Byte setzen
C011 LDA #$10       Bit 4 von
C013 BIT $01        Speicherstelle 1 testen
C015 BNE $C013      verzweige, wenn Bit gesetzt
C017 LDA $01        Speicherstelle 1 auslesen
C019 AND #$DF       Bit 5 löschen (Motor einschalten)
C01B STA $01        und zurückschreiben
C01D LDX #$00       Zähler für Zeitschleife
C01F DEY            Motor Zeit bis
C020 BNE $C01F      zum Erreichen der
C022 DEX            Arbeitsgeschwindigkeit
C023 BNE $C01F      geben
C025 LDA #$10       Bit 4 des ICR testen
C027 BIT $DC0D      Signal vom Band (HIGH/LOW-Wechsel)?
C02A BEQ $C027      verzweige, wenn nein
C02C DEX            schon 256 Signale empfangen?
C02D BNE $C027      verzweige, wenn nein
C02F STX $FB        Zeiger für Anzahl zu lesender Signale
C031 LDA #$18       auf $1800 (=6144) setzen
C033 STA $FC
C035 LDA #$19       Timer auf Startwert setzen und
C037 STA $DC0F      starten
C03A LDA #$10       Bit 4 des ICR testen
C03C BIT $DC0D      Signal vom Band?
C03F BEQ $C03C      verzweige, wenn nein
C041 LDA #$08       Timer anhalten
C043 STA $DC0F
C046 LDA $DC06      Timer LOW-Byte lesen
C049 LDX $DC07      Timer HIGH-Byte lesen
C04C LDY #$19       Timer wieder starten
C04E STY $DC0F
C051 EOR #$FF       Timer-Bits invertieren,
C053 TAY            um Anzahl der Taktzyklen
C054 TXA            seit letztem Signal
C055 EOR #$FF       zu erhalten
C057 TAX
C058 CPY #$C8       Timer-Wert mit
C05A SBC #$00       200 (=$C8) vergleichen
C05C BCC $C065      verzweige, wenn kleiner
C05E CPY #$58       Timer-Wert
```

```
C060 TXA          mit 600
C061 SBC #$02     (=$0258) vergleichen
C063 BCC $C068    verzweige, wenn kleiner
C065 JMP $FCE2    falls größer, Sprung nach RESET
C068 DEC $FB      Zähler für
C06A BNE $C03A    Anzahl zu lesender Signale
C06C DEC $FC      vermindern
C06E BNE $C03A    verzweige, wenn ungleich Null
C070 LDA #$08     Timer anhalten
C072 STA $DC0F
C075 LDA $D011    Bildschirm
C078 ORA #$10     wieder einschalten
C07A STA $D011
C07D STA $C0      Motorsteuer-Flag ungleich Null
C07F LDA $01      Speicherstelle Eins lesen
C081 ORA #$20     Bit 5 setzen (Motor anhalten)
C083 STA $01      Wert zurückschreiben
C085 CLI          Interrupts wieder zulassen
C086 RTS          Rücksprung
```

Dieses Programm demonstriert zum erstenmal die Benutzung der Timer. Der Programmbeginn ist der gleiche wie beim ersten Programm; alle Unterbrechungen werden ausgeschaltet, und der Motor wird eingeschaltet, sobald eine Recordertaste gedrückt wird (hier sollte es die PLAY-Taste sein). Außerdem läßt man dem Motor wieder Zeit zum Anlaufen.

Vorher ist aber schon der Startwert für den Timer auf $FFFF (=65535), also auf den Maximalwert, gesetzt worden. Um Zeiten zu messen, braucht man bloß den Zähler des Timers auszulesen und alle Bits dieses Wertes zu invertieren. Man erhält dann die abgelaufene Zeit plus Eins. Die 'plus Eins' können wir aber getrost vergessen, da es aufgrund der Ungenauigkeit der Datasette auf einen Taktzyklus mehr oder weniger nicht ankommt.

Sobald das Band läuft, werden erst einmal 256 Signale abgewartet, damit nicht die Gefahr besteht, daß zufälligerweise einige Signale, die gar nicht zur geschriebenen Signalfolge gehören, auf

ihren zeitlichen Abstand getestet werden. Solche Signale können beim Einschaltvorgang der Datasette unbeabsichtigt auftreten.

Das Erkennen eines Signals erfolgt folgendermaßen: Wir testen Bit 4 des Interruptkontrollregisters (ICR, $DC0D). Sobald dieses Bit den Wert Eins annimmt, lag ein Signal (eine negative Flanke) auf dem Band vor. Durch das Auslesen dieses Register wird automatisch das entsprechende Bit im ICR wieder gelöscht und man kann direkt auf das nächste Signal warten.

Jetzt kann die eigentliche Kopierschutzabfrage beginnen. Der Zähler $FB/$FC für die Anzahl der Signale wird auf $1800 (=6144) gesetzt. Wir haben zwar anfangs $2000 (=8192) Daten aufs Band gebracht, falls aber aus irgendwelchen Gründen der exakte Beginn der Aufnahme verpaßt werden sollte, ist hierdurch ein ausreichender Spielraum gegeben.

Als nächstes wird der Timer gestartet:

```
C035 LDA #$19       =%00011001
C037 STA $DC0F      ins Kontrollregister schreiben
```

```
Folgende Bits im Timer-Kontrollregister werden so gesetzt:

Bit 0: Timer wird gestartet.
Bit 3: Timer hält nach Ablauf an.
Bit 4: Startwert wird in den Zähler geladen.
```

Außerdem sind die Bits 5 und 6 auf Null, das bedeutet, der Timer zählt Systemtakte.

Sobald der Timer läuft, wird auf ein Signal vom Band gewartet. Kommt das Signal, halten wir den Timer an, lesen ihn aus und starten ihn neu. Dann invertieren wir den gelesenen Timer-Wert, um die vergangene Zeit in Taktzyklen zu erhalten. Diese Zeit betrug beim Schreiben 400 Zyklen und sollte demnach beim Lesen auch etwa diesen Wert betragen. In unserem Beispiel räu-

men wir dem Signal einen Spielraum von 200 Zyklen nach oben und unten ein. Sollten also zwei Signale in kürzerem Abstand als 200 Taktzyklen oder in längerem Abstand als 600 Taktzyklen aufeinander folgen, so gilt der Kopierschutz als nicht erkannt. In diesem Falle springt das Programm zur RESET-Routine, die den Rechner in den Einschaltzustand versetzt.

Sobald das Programm alle Signale richtig erkannt hat, macht es das gleiche wie die Schreibroutine bei ihrem Abschluß: Der Motor wird aus-, der Bildschirm eingeschaltet und Interrupts werden wieder zugelassen. Diese Abfrage wird also nur dann richtig beendet, falls der Kopierschutz erkannt wurde. Anderenfalls wird entweder ein RESET ausgelöst, oder, falls überhaupt keine Signale auf dem Band sind, die Warteschleife nicht verlassen.

Falls Sie dieses Programm selber als Kopierschutz verwenden wollen, so sollten Sie daran denken, daß jeder, der ebenfalls das Programm aus diesem Buch besitzt, ohne weiteres die Schutzaufzeichnung reproduzieren kann, wenn Sie das Programm unverändert übernehmen. Sie können allerdings auch Änderungen vornehmen, die dieses Problem ausschließen. Es bieten sich dabei zum Beispiel die Anzahl und der Abstand zwischen den Signalen an. Wie weit Sie bei solchen Änderungen gehen wollen, bleibt Ihnen selbst überlassen.

Nun zur Handhabung des Programms. Bringen Sie die Erkennungsroutine in dem zu schützenden Programm unter. Falls Sie ein BASIC-Programm schützen wollen, so können Sie das nachfolgende DATA-Listing dort einbauen. Zum Starten des Programms dient dann der Befehl 'SYS 49152'. Die Erkennungsroutine läßt sich aber auch leicht in ein Maschinenprogramm packen, da sie völlig relokatibel ist. Egal, mit welchem der beiden Fälle Sie es zu tun haben, Sie sollten Ihr Programm codieren und mit einem Autostart versehen, eventuell sogar gegen einen RESET schützen und die Erkennungsroutine nach ihrem Ablauf zerstören. Genügend Anregungen dazu finden Sie in den Kapiteln zum Programmschutz.

Nachdem sich Ihr Programm nun auf dem Band befindet, laden Sie (am besten von einer anderen Kassette) die Schreibroutine. Wenn diese Routine als DATA-Listing vorliegt, so starten Sie sie noch mit 'RUN'. Legen Sie nun wieder die Kassette mit Ihrem Programm ein, starten Sie die Schreibroutine mit 'SYS 49152' und betätigen Sie RECORD/PLAY am Recorder. Drei bis vier Sekunden später erscheint der Bildschirm wieder, und Sie sind fertig.

Spulen Sie nun die Kassette bis zum Anfang Ihres Programms zurück und laden Sie es. Falls Ihr Programm einen Autostart besitzt, geschieht alles weitere automatisch. Falls nicht, starten Sie es, aber lassen Sie am besten die PLAY-Taste gedrückt. Die Leseroutine sollte nun den Kopierschutz erkennen.

Hier nun die DATA-Listings der beiden Schutzprogramme:

Leseroutine: Bereich $C000 bis $C086

```
100 FORI=1TO135STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,AD,11,D0,29,EF,8D,11,D0,A9,FF,8D,06,DC,8D, 48
301 DATA07,DC,A9,10,24,01,D0,FC,A5,01,29,DF,85,01,A2, 99
302 DATA00,88,D0,FD,CA,D0,FA,A9,10,2C,0D,DC,F0,FB,CA, 108
303 DATAD0,F8,86,FB,A9,18,85,FC,A9,19,8D,0F,DC,A9,10, 126
304 DATA2C,0D,DC,F0,FB,A9,08,8D,0F,DC,AD,06,DC,AE,07, 109
305 DATADC,A0,19,8C,0F,DC,49,FF,A8,8A,49,FF,AA,C0,C8, 0
306 DATAE9,00,90,07,C0,58,8A,E9,02,90,03,4C,E2,FC,C6, 144
307 DATAFB,D0,CE,C6,FC,D0,CA,A9,08,8D,0F,DC,AD,11,D0, 172
308 DATA09,10,8D,11,D0,85,C0,A5,01,09,20,85,01,58,60, 217
```

Schreibroutine: Bereich $C000 bis $C057

```
100 FORI=1TO88STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,AD,11,D0,29,EF,8D,11,D0,A9,10,24,01,D0,FC, 54
301 DATAA5,01,29,DF,85,01,A9,04,A2,00,88,D0,FD,CA,D0, 114
302 DATAFA,38,E9,01,D0,F5,86,FB,A9,20,85,FC,A5,01,29, 123
303 DATAF7,85,01,A2,14,CA,D0,FD,09,08,85,01,A2,3C,CA, 9
304 DATAD0,FD,C6,FB,D0,E8,C6,FC,D0,E4,AD,11,D0,09,10, 99
305 DATA8D,11,D0,A5,01,09,20,85,01,85,C0,58,60,A4,A4, 8
```

Diese Art des Schutzes ist zwar an sich schon sehr wirkungsvoll, birgt aber noch den Nachteil in sich, daß sie die geringe Ladegeschwindigkeit beibehält. Unser nächstes Ziel wird demnach sein, ein eigenes, schnelleres Aufzeichnungsformat zu entwickeln.

Ganz zu Anfang haben wir die Behauptung aufgestellt, es lasse sich kein allgemeines Kopierprogramm schreiben. Jetzt wird es Zeit, diese Behauptung zu rechtfertigen. Aufgezeichnete Daten bestehen aus Signalen, die sich in unterschiedlichen Zeitabständen auf dem Band befinden. Da ein Kopierprogramm nicht "weiß" (und aufgrund der Gleichlaufschwankungen auch nicht feststellen kann), ob dieses Aufzeichnungsformat zwei oder mehr unterschiedliche Signalabstände verwendet, kann es prinzipiell nur eines machen, nämlich jedesmal die Zeit von einem Signal bis zum nächsten zu messen und abzuspeichern. Diese Methode benötigt zwangsläufig wesentlich mehr Speicherplatz, als das Programm nach dem normalen Einladen wirklich belegt. Ein Beispiel: Ein Aufzeichnungsformat arbeitet mit zwei verschieden Zeiten für ein Null- und ein Eins-Bit. Zusätzlich wird noch nach jedem achten Bit ein weiteres Signal (mit anderen möglichen Zeiten) geschrieben. Für eine gute Zeitmessung benötigt man pro Signal zwei Bytes. Wenn das zu kopierende Programm vier Kilobyte belegt, würden demnach 4096 * (8+1) = 36864 Signale auf dem Band sein. Das Kopierprogramm würde dem-

nach 2 * 36864 = 73728 Byte an Speicherplatz benötigen, also wesentlich mehr, als der C64 zur Verfügung hat. Man kann die Aufnahme auch nicht in mehrere Teile aufsplitten, da eine zusammenhängende Datenfolge auch zusammenhängend geschrieben werden muß.

Man könnte annehmen, daß es eventuell sinnvoll wäre, ein Kopierprogramm für sehr kurze Programme zu schreiben oder eine Speichererweiterung zu verwenden. Sicherlich könnte man so tatsächlich Kopien anfertigen, die allerdings das gleiche Problem beinhalten würden, als wenn man mit einem Überspielkabel arbeiten würde. Schließlich sind die gelesenen Zeitabstände wegen der vielzitierten Gleichlaufschwankungen nicht exakt die gleichen wie die geschriebenen. Daher wird jede weitere Kopie ein wenig schlechter sein wie ihr Vorgänger.

Um also sinnvoll Kopien anfertigen zu können, muß das Kopierprogramm die Lese- und Schreibzeiten des jeweiligen Formates kennen und kann demnach nicht mehr für alle möglichen Formate verwendbar sein.

5.5.3 Ein neues Aufzeichnungsformat am Beispiel des "KKS"

Wir haben uns zum Ziel gesetzt, in diesem Kapitel ein eigenes Aufzeichnungsformat für Bandaufnahmen zu entwickeln. Dieses Programm soll den Namen "KKS" (Kassetten-Kopierschutz-System) tragen. Das KKS muß folgende Eigenschaften besitzen:

1. nicht mit diversen Kopierprogrammen zu kopieren
2. höhere Aufzeichnungsgeschwindigkeit
3. automatisches Starten der schnellen Laderoutine und automatisches Starten des nachgeladenen Programms.
4. einfache Handhabung für Leser, die sich nicht im Detail mit dem Programm auseinandersetzen wollen.

5. speichern aus dem RAM-Bereich unter dem BASIC-Interpreter

6. Option zum Laden in andere Speicherbereiche, als die, aus denen abgespeichert wurde.

Prinzipiell sieht das Programm so aus: Zuerst wird der Vektor der SAVE-Routine des Betriebssystems auf die eigene Routine umgestellt. Gibt man jetzt den Befehl 'SAVE "Programmname"' ein, wird zuerst ein Teil der Laderoutine nach $02A8 (=680) geschoben. Der andere Teil befindet sich im Filenamen und gelangt demnach beim Einladen in den Kassettenpuffer. Die Laderoutine wird dann ganz normal, "langsam" und inklusive Autostart abgespeichert. Danach wird das eigentliche Programm unter dem neuen Format abgelegt.

Beim Einladen gibt man wie gewohnt 'LOAD' ein und betätigt die PLAY-Taste. Das Betriebssystem lädt und startet die neue Laderoutine automatisch. Diese lädt und startet nun das Hauptprogramm.

Wie sieht das neue Aufzeichnungsformat aus? Zuerst wird eine sogenannte "Synchronisations-Markierung" geschrieben, die zum Erkennen des Programmanfangs dient. Genaueres dazu erfahren Sie im übernächsten Absatz. Als nächstes folgen die Start- und Endadresse des zu ladenden Programms. Danach steht dann das Hauptprogramm. Ganz am Ende kommt noch ein Byte als Prüfsumme, welches das Erkennen von Ladefehlern ermöglicht. Die Prüfsumme wird gebildet, indem alle Bytes mit dem Befehl 'EOR' verknüpft werden. Wenn die abgespeicherte Prüfsumme mit der beim Laden gebildeten nicht übereinstimmt, können die geladenen Daten nicht die gleichen sein wie die ursprünglich abgespeicherten.

Der oben beschriebene Vorgang setzt voraus, daß die Möglichkeit besteht, Bits und Bytes aufs Band zu bringen. Wir gehen dabei so vor, daß ein Eins-Bit als langer und ein Null-Bit als kurzer Abstand des vorhergehenden Signals vom nächsten codiert wird. Die hier verwendeten Zeiten sind 388 Taktzyklen

für die Null und 643 Zyklen für die Eins. Dementsprechend werden dann je acht gelesene Bits zu einem Byte zusammengefaßt.

Die Laderoutine steht zunächst einmal vor dem Problem, erkennen zu müssen, welches das erste Bit ist, das zu den aufgezeichneten Daten gehört. Dieses Problem wird mit Hilfe der schon angesprochenen Synchronisationsmarkierung gelöst. Diese Markierung besteht aus einer Folge gleicher Bytes, deren Bitkombination so beschaffen ist, daß es eindeutig ist, welches Bit an welcher Stelle eines Bytes steht. Beispielsweise ist eine Folge von $55-Bytes ($55=%01010101) dafür nicht geeignet, da die Bitkombination mehrerer hintereinander liegender Bytes so aussieht:

```
...0101010101010101010101010101...
```

Hier ist nicht unbedingt klar, daß das erste Bit der Folge tatsächlich das erste Bit in einem Byte der Sync-Markierung ist. Es könnte genauso gut das dritte oder das fünfte Bit sein. Dagegen schließt zum Beispiel eine $03-Byte-Folge ($03=%00000011) alle Mehrdeutigkeiten aus:

```
...011000000110000001100000011000...
```

Diese Folge kann nur auf eine Weise wieder in $03-Bytes zerlegt werden:

```
...011 00000011 00000011 00000011 000...
```

Die Synchronisationsmarkierung wird meistens noch durch ein oder mehrere Bytes ergänzt, welche die Wahrscheinlichkeit verringern, eine solche Markierung mit einer Bytefolge einer anderen Bandaufzeichnung zu verwechseln. Bei unserem Beispiel verwenden wir ein Byte mit dem Wert $48.

Wie erreicht man nun das genaue Timing beim Schreiben und Lesen? Am einfachsten geht es mit einer Interruptroutine. Beim Schreiben wird der CIA-Timer als Interruptquelle verwendet. Dieser Timer bleibt auf einem konstanten Wert. Beim Auslösen eines Interrupts wird getestet, ob das nächste zu schreibende Bit

ein Null-Bit ist. Falls ja, so wird sofort ein Signal auf dem Band erzeugt, falls nein, so wird vorher noch eine Zeitschleife ausgeführt. Danach startet man den Timer neu.

Beim Lesen wird ebenfalls der Timer benutzt. Diesmal aber erzeugt nicht er einen IRQ, sondern das Signal von der Kassette. Das Bit, welches den Ablauf des Timer-Zählers anzeigt, wird damit gleichzeitig zum gelesenen Bit, da es genau dann eins wird, wenn der Abstand zweier Signale einen bestimmten Wert überschreitet.

Der Zeitablauf beim Lesen und Schreiben errechnet sich beim KKS wie folgt: Bei Verwendung des IRQ-Vektors $0314/$0315 liegt zwischen dem Auslösen des Interrupts und dem Erreichen der eigenen Interruptroutine eine Zeitspanne von durchschnittlich 39 Taktzyklen. Bei der Schreibroutine vergehen bei einem Null-Bit noch 16 Zyklen, bis das Signal auf die Leitung gegeben wird, und noch einmal sechs Zyklen, bis der Timer neu gestartet wird. Wenn man den Timer auf einen Wert von 327 (=$0147) Zyklen setzt, so ergibt sich die Zeitdauer für ein Null-Signal folgendermaßen:

Starten des Timers:	6 Zyklen
Zeit bis zum IRQ :	327 Zyklen
Zeit bis zum Erreichen der eigenen IRQ-Routine:	39 Zyklen
Zeit bis zum Ändern der Signal-Leitung:	16 Zyklen
Gesamtzeit:	388 Zyklen

Für ein Eins-Bit kommen zusätzlich durch die Zeitschleife noch einmal 255 Taktzyklen hinzu, wodurch sich der Abstand zum vorhergehenden Signal auf 255 + 388 = 643 Taktzyklen erhöht.

Egal, ob ein Null- oder Eins-Bit geschrieben wird, die Schreibleitung bleibt für ca. 80 Taktzyklen auf LOW, was aber für die eigentliche Zeitberechnung unwichtig ist.

Beim Einlesen der Daten muß der Timer so eingestellt werden, daß er bei einem kurzen Bandsignal in der Zeit bis zum Auslesen des Interrupt-Kontroll-Registers mit Sicherheit noch nicht abgelaufen ist, dagegen bei einem langen Bandsignal das Timer-Bit im ICR mit Sicherheit auf Eins steht. Für ein Null-Bit läßt sich folgende Zeitberechnng anstellen:

Beginn: negative Flanke löst Interrupt aus

Zeit bis zum Erreichen der Interrupt-Routine:	39 Zyklen
Zeit zum Lesen des ICR:	4 Zyklen
Zeit bis zum Start des Timers:	6 Zyklen
Gesamtzeit:	49 Zyklen

Zeit bis zum nächsten Signal:	388 - 49 = 339 Zyklen
Zeit bis zum Lesen des ICR:	39 + 4 = 43 Zyklen
Gesamtzeit:	382 Zyklen

Die gleiche Rechnung für ein Eins-Bit liefert als Ergebnis 637 Zyklen. Der Startwert für den Timer sollte demnach möglichst genau zwischen diesen beiden Werten liegen, also bei 510 (=$01FE) Zyklen.

Wie gelangen jetzt die Bytes aus dem Speicher auf das Band? Die Interruptroutine für das Abspeichern schreibt immer genau ein Byte. Dazu benutzt sie vier Adressen:

- ein Schieberegister, aus welchem immer das nächste zu schreibende Bit geschoben wird
- ein Bitzähler, der festhält, wieviele Bits schon aus dem Schieberegister herausgeschoben wurden
- ein Zwischenspeicher für das nächste Byte, welches ins Schieberegister übernommen wird
- ein Speicher, der anzeigt, ob der Zwischenspeicher schon von der Interruptroutine ausgelesen wurde

Das neben der Interruptroutine laufende Hauptprogramm wartet, bis die Interruptroutine das vorherige Byte aus dem Zwischenspeicher geholt hat, und schreibt dann das nächste Datenbyte dorthin. Der Speicher, an dem das Hauptprogramm erkennt, daß der Zwischenspeicher frei ist, wird zunächst von der Interruptroutine auf $80 (=128) gesetzt. Wenn das Hauptprogramm den Zwischenspeicher wieder belegt hat, ersetzt es die 128 durch eine Null. Sobald durch die Interruptroutine die letzten acht Bits aus dem Schieberegister auf das Band gebracht wurden, holt sie das Byte aus dem Zwischenspeicher ins Schieberegister, wodurch sich der Kreis schließt.

Die Lese-Interruptroutine arbeitet genau umgekehrt, wobei wieder die gleichen vier Register verwendet werden. Jedes eingelesene Bit gelangt zunächst ins Schieberegister. Jeweils nach acht Bits wird der Inhalt des Schieberegisters in den Zwischenspeicher übertragen. Die Übergabe der Bytes an das Hauptprogramm vollzieht sich genauso wie die Übergabe von Bytes beim Schreiben. Der einzige noch zu betrachtende wichtige Fall ist die Synchronisation, also das Erkennen, bei welchem Signal die gespeicherten Daten anfangen. Hierzu läßt das Hauptprogramm die Interruptroutine arbeiten, ohne Daten aus dem Zwischenspeicher zu übernehmen. Es fragt stattdessen das Schieberegister ab, ob es den Wert $03 (von der Synchronisationsmarkierung) annimmt. Sobald dieser Fall eintritt, wird noch vor der nächsten IRQ-Anforderung der Zähler für die Anzahl der gelesenen Bits auf den Maximalwert sieben gesetzt. (Der Zähler wird immer von sieben auf null heruntergezählt.) Jetzt können regulär Bytes über den Zwischenspeicher eingelesen werden.

Zu Beginn dieses Unterkapitels haben wir sechs Bedingungen an unser KKS gestellt, auf deren Erfüllung wir nun zu sprechen kommen. Der erste Punkt, die schwere Kopierbarkeit, wird allein dadurch erreicht, daß wir ein neues Aufzeichnungsformat benutzen.

Der zweite Punkt, die höhere Aufzeichnungsgeschwindigkeit, ergibt sich durch die gewählten Zeiten zwischen den Signalen. Die durchschnittliche Zeitspanne beträgt hier 516 Taktzyklen.

Bei einer Taktfrequenz von 0.98 Megahertz ergibt sich eine Aufzeichnungsrate von etwa 1900 Bit/Sekunde. Das ist mehr als sechsmal so schnell wie bei dem Aufzeichnungsformat des Betriebssystems. Wenn Sie wollen, können Sie die Geschwindigkeit noch erhöhen. Die nötigen Änderungen dazu sind im nachfolgend abgedruckten Assemblerlisting an drei Stellen durchzuführen:

$C0C9 Startwert des Timers beim Schreiben
$C148 Zeitschleife für Differenz zwischen null und eins
$037D Startwert des Timers beim Lesen

Bedenken Sie aber, daß eine höhere Geschwindigkeit immer eine Verringerung der Datensicherheit zur Folge hat und daß sich die Aufzeichnungsrate wegen der Gleichlaufschwankungen der Datasette nicht beliebig steigern läßt. Sollten Sie beispielsweise keine Datasette, sondern einen Hi-Fi-Rekorder mit einem Interface zum Anschluß an den C64 besitzen, so können Sie die Geschwindigkeit wesentlich höher ansetzen.

Das automatische Starten der neuen Laderoutine wird durch Ändern des BASIC-Eingabe-Vektors $0302/$0303 (=770/771) erreicht. Das automatische Starten des Hauptprogramms läßt sich durch Aufruf der BASIC-Interpreterroutine RUN realisieren, wie es schon in Teil 3 beschrieben wurde:

```
LDA #$00
JSR $A871
JMP $A7AE
```

Soll Ihr Programm nicht durch den Befehl RUN, sondern durch den Befehl SYS gestartet werden, so ersetzen Sie einfach diesen Aufruf durch einen direkten Einsprung mit dem Befehl 'JMP'.

Das Umstellen des SAVE-Vektors macht die Handhabung des KKS relativ einfach. Nicht nur der BASIC-Befehl 'SAVE' arbeitet mit dem Programm zusammen, sondern auch diverse Maschinensprachemonitore.

In diesem Zusammenhang können wir jetzt auch auf die Bedienung des KKS zu sprechen kommen. Tippen Sie einfach den nachfolgend abgedruckten BASIC-Lader ab und geben Sie 'RUN' ein. Nach Ablauf dieses Programms muß das KKS mit 'SYS 49152' gestartet werden. Möchten Sie eines Ihrer eigenen Programme schützen, laden Sie es ein und speichern Sie es ganz normal mit 'SAVE "Programmname"' ab. Wenn Sie nicht jedesmal den BASIC-Lader einladen und starten wollen, um das Kopierschutz-System benutzen zu können, so können Sie das Maschinenprogramm auch nach Ablauf des Laders direkt abspeichern. Entweder verwenden Sie dazu einen Maschinensprache-Monitor oder folgende Befehlszeilen:

```
POKE 43,0:POKE 44,192:POKE 45,106:POKE 46,194 :POKE 56,208 :CLR
SAVE"KKS",1,1
```

Jetzt können Sie das KKS mit 'LOAD "KKS"' laden und sofort mit dem obigen SYS-Befehl installieren. Es empfiehlt sich, danach 'NEW' einzugeben, damit alle BASIC-Zeiger auf einen sinnvollen Wert gesetzt werden. (Das Maschinenprogramm wird dadurch selbstverständlich nicht gelöscht.) Achtung: Das Programm verstellt den SAVE-Vektor, der durch Betätigen von STOP/RESTORE wieder auf seinen alten Wert zurückgesetzt wird. Sollten Sie also diese Tastenkombination benutzt haben, müssen Sie erst noch einmal 'SYS 49152' eingeben, um das KKS wieder verwenden zu können.

Um Programme aus dem RAM-Bereich unter dem BASIC-Interpreter abzuspeichern, schreibt das KKS vor Beginn des Speichervorgangs den Wert $06 in die Speicherstelle 1. Hinterher erhält diese Speicherstelle wieder ihren ursprünglichen Wert. Haben Sie ein Programm in den Speicher geladen, dessen Länge 38 KBytes (154 Blöcke auf einer Diskette) überschreitet, und wollen Sie es nun mit dem KKS abspeichern, so klappt das nur, wenn Sie keinen Filenamen angeben. Anderenfalls erhalten Sie die Fehlermeldung 'OUT OF MEMORY ERROR'. Hier kann man sich mit einem Trick helfen, indem man nämlich 'POKE 56,208:CLR' eingibt. Dadurch kann der BASIC-Interpreter seine Zeichenketten in einem freien Speicherbereich ablegen. Wollen

Sie nach dem Abspeichervorgang ein BASIC-Programm ablaufen lassen, dann bringen Sie die Speicherstelle 56 wieder auf ihren alten Wert: 'POKE 56,160:CLR'.

Der sechste der genannten Punkte, das Laden in andere Speicherbereiche als die, aus denen die Daten abgespeichert wurden, läßt sich durch eine einfache Änderung am KKS verwirklichen. Da das KKS einen Unterschied zwischen der Ladeanfangsadresse und der Anfangsadresse beim Abspeichern macht, kann man diesen Adressen auch unterschiedliche Werte geben. Dazu haben wir im KKS an zwei Stellen durch 'NOP'-Befehle Platz gehalten. Um also eine andere Ladeadresse zu erhalten, ersetzen Sie mit einem Maschinensprachemonitor die "NOP's" durch folgende Befehle:

```
C07E LDA #$HIGH-Byte der Ladeadresse
C085 LDA #$LOW-Byte der Ladeadresse
```

Diese Änderung ist in zweierlei Hinsicht nützlich. Einmal lassen sich damit Programmteile, die in Bereiche geladen werden sollen, aus denen man nichts abspeichern kann, trotzdem abspeichern. Zu diesen Speicherbereichen zählen der Bereich $C000 bis $C269 (weil dort das KKS selbst liegt) und der Bereich $E000 bis $FFFF (weil dort das Betriebssystem liegt). Da die Laderoutine komplett im Kassettenpuffer liegt und oberhalb von $0400 (=1024) keinen Speicherplatz beansprucht, kann man problemlos in diese Bereiche Programme laden. Haben Sie beispielsweise ein Programm, das im Bereich von $C000 bis $CFFF liegt, so verschieben Sie dieses Programm zuerst in einen anderen Bereich, zum Beispiel $2000 bis $2FFF (am besten mit einem Monitor). Laden Sie dann das KKS und nehmen folgende Änderung vor:

```
C07E LDA #$C0      HIGH-Byte von $C000
C085 LDA #$00      LOW-Byte von $C000
```

Als nächstes starten Sie das KKS mit 'SYS 49152'. Jetzt können Sie das Programm von $2000 bis $2FFF abspeichern. Es wird dann so eingeladen, als hätten Sie es aus dem Bereich $C000 abgespeichert.

Es war davon die Rede, daß die gerade besprochene Änderungsoption noch zu einem anderen Zweck genutzt werden kann. Man kann damit nämlich einen interessanten Programmschutz aufbauen, der sich normalerweise nur bei einem Diskettenprogramm realisieren läßt. Die Speicherstellen $AC/$AD werden beim Einladen als Zeiger auf das jeweils nächste Byte verwendet (siehe Assemblerlisting). Wenn man dort Werte hineinlädt, gelangen die nachfolgenden Bytes an eine völlig andere Stelle. Vorsicht ist dabei geboten, wenn man durch die Speicherstelle $9B hindurchlädt. Sie enthält nämlich die Prüfsumme über die geladenen Daten. Falls diese bei Beendigung des Ladevorgangs nicht stimmt, so nimmt das KKS an, die Daten seien fehlerhaft, und löst einen RESET aus.

Legen Sie beispielsweise folgende Bytefolge im Speicher ab:

```
109B 11 00 00 00 00 00 00 00
10A3 00 00 00 00 00 00 00 00
10AB 00 AC 7F 00 00 00 00 00

1100 74 A4 74 A4 C3 C2 CD 38
1108 30 00 00 00 00 00 00 00
```

Speichern Sie dann den Bereich von $109B bis $1109 so ab, daß er nach $009B geladen wird. Beim Einladen wird zuerst die Prüfsumme zerstört, was hier aber durchaus beabsichtigt ist. Die folgenden Null-Bytes bewirken gar nichts. Erst das Byte $AC, welches nach $00AC kommt, ist von Bedeutung. Das KKS schreibt also den Wert $AC nach $00AC und inkrementiert dann diesen Zeiger. Das nächste Byte, $7F, kommt dann nach $00AD, wodurch der Zeiger nach dem nächsten Inkrementieren auf $7FAE steht. Dorthin werden dann alle folgenden Bytes geladen. In $8000 steht dann die bekannte 'CBM80'-Erkennung. (Genaueres dazu finden Sie unter dem Kapitel "Programmschutz".) Sobald der Ladevorgang abgeschlossen ist, prüft das KKS die Korrektheit der Prüfsumme. Da diese aber nicht mehr stimmt, kommt es zu einem RESET. Durch das 'CBM80' springt der

Prozessor aber nach $A474, also in den BASIC-Interpreter. Sie können diese Einsprungadresse selbstverständlich so umändern, daß dadurch Ihr eigenes Programm gestartet wird.

Wir wollen hier nicht auf weitere Details dieses Programmschutzes eingehen, da es sich an dieser Stelle nur um eine Anregung handeln soll. Sie finden allerdings eine ähnliche Version in dem Programmschutzteil dieses Buches, die sich dort aber nur auf die Diskettenstation bezieht.

Damit wären wir schon fast am Ende dieses Kapitels angelangt. Es folgen nur noch die beiden Programmlistings und eine Auflistung der in diesem Zusammenhang interessanten Systemadressen und -routinen. Wir hoffen, Ihnen hiermit ein Werkzeug an die Hand gegeben zu haben, mit dessen Hilfe Sie sich selbst einen Kopierschutz sozusagen "nach Maß" zusammenstellen können. Vielleicht schafft es ja eines Tages doch jemand, den "unkopierbaren" Kopierschutz zu entwickeln.

Das Kassetten-Kopierschutz-System KKS (Assemblerlisting)

verwendete Adressen:

```
$9B     : Prüfsumme des geladenen/gespeicherten Programms
$9C     : wenn negativ: Byte wurde geschrieben/gelesen
$AC/$AD: Zeiger auf aktuelles Byte
$AE/$AF: Zeiger auf Endadresse+1
$B4     : Zähler für Anzahl gelesener/geschriebener Bits
$BE     : aktuelles Byte
$BF     : Schieberegister für zu schreibendes/lesendes Byte
$C1/$C2: geschriebene Ladeanfangsadresse
```

Assemblerlisting:

SAVE-Routine:

```
C000 LDA #$0B       SAVE-Vektor ändern: LOW-Byte von $C00B
C002 STA $0332      nach $0332: LOW-Byte des SAVE-Vektors
C005 LDA #$C0       HIGH-Byte von $C00B
C007 STA $0333      nach $0333: HIGH-Byte des SAVE-Vektors
C00A RTS            Rücksprung
C00B LDA $BA        Geräteadresse
C00D CMP #$01       =1 (Kassette)?
C00F BEQ $C014      wenn ja, weiter
C011 JMP $F5ED      wenn nein, normale SAVE-Routine
C014 LDX #$00       Verschiebeschleife vorbereiten
C016 LDA $C23E,X    Teil der Laderoutine
C019 STA $02A8,X    nach $02A8 verschieben
C01C INX            Zähler erhöhen
C01D CPX #$48       schon alle Bytes verschoben?
C01F BNE $C016      verzweige, wenn nein
C021 LDY #$00       Zeiger auf Filenamen vorbereiten
C023 CPY $B7        schon kompletten Filenamen verschoben?
C025 BEQ $C031      verzweige, wenn ja
C027 LDA ($BB),Y    Filenamensbyte holen
C029 STA $C183,Y    und vor Ladeprogramm setzen
C02C INY            Zeiger erhöhen
C02D CPY #$10       schon 16 Zeichen verschoben
C02F BNE $C023      verzweige, wenn nein
C031 CPY #$10       schon 16 Zeichen verschoben?
C033 BEQ $C03E      verzweige, wenn ja
C035 LDA #$20       wenn nein, mit Leerzeichen auffüllen
C037 STA $C183,Y    Leerzeichen hinter Filenamen setzen
C03A INY            Zeiger erhöhen
C03B JMP $C031      unbedingter Sprung
C03E LDA #$01       Filenummer
C040 TAX            Geräteadresse=1
C041 TAY            Sekundäradresse=1 (nicht verschieblich)
C042 JSR $FFBA      FILPAR
C045 LDA #$BC       Filenamenlänge incl. Laderoutine
C047 LDX #$83       Filenamenbeginn
C049 LDY #$C1       bei $C183
```

```
C04B JSR $FFBD      FILNAM
C04E LDA $AE        LOW-Byte Endadresse
C050 PHA            auf Stapel retten
C051 LDA $AF        HIGH-Byte Endadresse
C053 PHA            retten
C054 LDA $C1        LOW-Byte Anfangsadresse
C056 PHA            retten
C057 LDA $C2        HIGH-Byte Anfangsadresse
C059 PHA            retten
C05A LDA #$A8       $02A8
C05C STA $C1        als neue Anfangsadresse
C05E LDA #$02       nach $C1/$C2
C060 STA $C2
C062 LDA #$04       $0304
C064 STA $AE        als neue Endadresse
C066 LDA #$03       nach $AE/$AF
C068 STA $AF
C06A LDA #$D3       Startadresse
C06C STA $0302      für Autostart
C06F LDA #$02       nach $0302/$0303 bringen
C071 STA $0303      (BASIC-Vektor für Eingabe einer Zeile)
C074 LDA #$00       0 nach $9D bringen (verhindert Anzeige
C076 STA $9D        des verlängerten Filenamens)
C078 JSR $F5ED      normale SAVE-Routine aufrufen
C07B PLA            HIGH-Byte Anfangsadresse
C07C STA $AD        zurückholen
C07E NOP               (Platz für Änderungen)
C07F NOP               (Platz für Änderungen)
C080 STA $C2        gleich HIGH-Byte für Ladeanfangsadresse
C082 PLA            LOW-Byte Anfangsadresse
C083 STA $AC        zurückholen
C085 NOP               (Platz für Änderungen)
C086 NOP               (Platz für Änderungen)
C087 STA $C1        gleich LOW-Byte für Ladeanfangsadresse
C089 PLA            HIGH-Byte Endadresse
C08A STA $AF        zurückholen
C08C PLA            LOW-Byte Endadresse
C08D STA $AE        zurückholen
C08F JSR $E453      BASIC-Vektoren wieder zurücksetzen
C092 SEI            Interrupt verhindern
```

```
C093 LDA $D011      Unterbrechungen vom VIC
C096 AND #$EF       durch Abschalten des Bildschirms
C098 STA $D011      unterbinden
C09B LDA #$44       Interruptvektor
C09D STA $0314      auf $C144
C0A0 LDA #$C1       für SAVE-Routine
C0A2 STA $0315      verbiegen
C0A5 LDA $01        Speicherkonfiguration
C0A7 PHA            auf Stapel retten
C0A8 LDA #$06       eigene Konfiguration einstellen
C0AA STA $01        dabei gleichzeitig Datasettenmotor an
C0AC STA $C0        Motorsteuerflag ungleich Null
C0AE LDA #$03       Zeitschleife vorbereiten
C0B0 LDY #$00
C0B2 STY $9B        Prüfsumme auf 0
C0B4 DEX            Zeitschleife für Hochlaufzeit
C0B5 BNE $C0B4      des Motors
C0B7 DEY            (dreimal
C0B8 BNE $C0B4      so lang
C0BA SEC            wie
C0BB SBC #$01       beim
C0BD BNE $C0B4      Laden)
C0BF LDA #$7F       alle Interruptmöglichkeiten der CIA 1
C0C1 STA $DC0D      abschalten
C0C4 LDA #$82       Interrupt durch Timer B
C0C6 STA $DC0D      zulassen
C0C9 LDA #$47       Timer B
C0CB STA $DC06      auf Wert
C0CE INX            $0147
C0CF STX $DC07      stellen
C0D2 LDA #$07       Bitzähler auf
C0D4 STA $B4        Anfangswert sieben stellen
C0D6 LDA #$19       Timer starten
C0D8 STA $DC0F
C0DB LDA $DC0D      Interruptregister löschen
C0DE CLI            Interrupts zulassen
C0DF LDA #$03       Wert Drei zur Synchronisation
C0E1 JSR $C17A      auf Band schreiben
C0E4 DEY            schon 256mal geschrieben?
C0E5 BNE $C0DF      verzweige, wenn nein
```

```
C0E7 LDA #$48       $48 als Prgrammbeginnkennzeichnung
C0E9 JSR $C17A      auf Band schreiben
C0EC LDA $C1        LOW-Byte Anfangsadresse
C0EE JSR $C17A      auf Band schreiben
C0F1 LDA $C2        HIGH-Byte Anfangsadresse
C0F3 JSR $C17A      auf Band schreiben
C0F6 LDA $AE        LOW-Byte Endadresse im Speicher
C0F8 SEC
C0F9 SBC $AC        minus LOW-Byte Anfangsadr. im Speicher
C0FB PHA            retten
C0FC LDA $AF        HIGH-Byte Endadresse im Speicher
C0FE SBC $AD        minus HIGH-Byte Endadresse im Speicher
C100 TAX            retten
C101 PLA            LOW-Byte Programmlänge
C102 CLC
C103 ADC $C1        plus LOW-Byte Anfangsadr. für Ladeprg.
C105 JSR $C17A      auf Band schreiben
C108 TXA            HIGH-Byte Programmlänge
C109 ADC $C2        plus HIGH-Byte Anfangsadr. für Ladeprg.
C10B JSR $C17A      auf Band schreiben
C10E LDA ($AC),Y    Programm-Byte holen (Y=0)
C110 TAX            zwischenspeichern
C111 EOR $9B        Prüfsumme bilden
C113 STA $9B        und wegspeichern
C115 TXA            Programm-Byte zurückholen
C116 JSR $C17A      auf Band schreiben
C119 JSR $FCDB      laufender Prg.-zeiger $AC/$AD erhöhen
C11C JSR $FCD1      schon Endadresse $AE/$AF erreicht?
C11F BNE $C10E      verzweige, wenn nein
C121 LDA $9B        Prüfsumme holen
C123 JSR $C17A      auf Band schreiben
C126 JSR $C17A      warten, bis letztes Byte
C129 JSR $C17A      vollständig geschrieben worden ist
C12C SEI            Interrupt sperren
C12D PLA            Speicherkonfigurationswert holen
C12E ORA #$20       Motor aus
C130 STA $01        alte Konfiguration wieder herstellen
C132 JSR $FD15      Interruptvektor zurücksetzen
C135 JSR $FDA3      CIAs wieder auf alten Interruptbetrieb
C138 JSR $FC93      Motor aus, Bildschirm an
```

```
C13B LDA #$0E       Randfarbe des Bildschirms
C13D STA $D020      wieder auf Standartwert
C140 CLI            Interrupts wieder zulassen
C141 JMP $A474      zurück zum BASIC-Interpreter
```

Interruptroutine für Band schreiben:

```
C144 ASL $BF        Bit aus Schieberegister holen
C146 BCC $C14D      verzweige, wenn Bit gleich Null
C148 LDX #$33       Zeitschleife:
C14A DEX            ca. $33*5 (=255) Taktzyklen warten
C14B BNE $C14A      verzweige, wenn noch nicht fertig
C14D LDA $01        Portbyte holen
C14F AND #$F7       Bit 3 auf Null
C151 STA $01        Signal auf Band schreiben
C153 LDX #$19       Timer neu starten
C155 STX $DC0F
C158 LDX #$10       Zeitschleife:
C15A DEX            ca. $10*5 (=80) Taktzyklen warten
C15B BNE $C15A      verzweige, wenn noch nicht fertig
C15D ORA #$08       Bit 3 auf Eins
C15F STA $01        Signal wieder zurücksetzen
C161 INC $D020      Kontrolle durch Ändern der Randfarbe
C164 DEC $B4        schon komplettes Byte geschrieben?
C166 BPL $C174      verzweige, wenn nein
C168 LDA #$07       Bitzähler wieder auf 7
C16A STA $B4        setzen
C16C LDA $BE        nächstes zu schreibendes Byte
C16E STA $BF        ins Schieberegister
C170 LDA #$80       Flag für Byte geholt
C172 STA $9C        setzen
C174 LDA $DC0D      Interrupt-Port freimachen
C177 JMP $FEBC      Interrupt-Ende
```

Ein Byte schreiben

```
C17A BIT $9C        letztes Byte schon geholt?
C17C BPL $C17A      verzweige, wenn nein
```

```
C17E STA $BE        nächstes Byte übergeben
C180 STY $9C        'Byte geholt'-Signal löschen
C182 RTS            Rücksprung

C183 bis C192: Platz für Filenamen
```

Die nun folgende Laderoutine liegt im Speicher ab $C193, wird aber beim Einladen nach $0351 verschoben.

```
0351 SEI            Interrupts verhindern
0352 LDA #$0B       Unterbrechungen vom VIC durch
0354 STA $D011      Abschalten des Bildschirms verhindern
0357 LDA #$A8       Interruptvektor
0359 STA $0314      auf $02A8
035C LDA #$02       für Laderoutine
035E STA $0315      verbiegen
0361 LDA $01        Port-Byte holen
0363 AND #$1F       Bit für Datasettenmotor auf Nnull
0365 STA $01        Motor starten
0367 STA $C0        Motorsteuerflag ungleich Null
0369 LDY #$00       Zeitschleife vorbereiten
036B STY $9B        Prüfsumme auf Null
036D DEX            Zeitschleife, um
036E BNE $036D      Motor Zeit
0370 DEY            zum Anlaufen
0371 BNE $036D      zu geben
0373 LDA #$7F       Interrupts durch CIA 1
0375 STA $DC0D      verhindern
0378 LDA #$90       Interrupt durch Signal am Pin FLAG
037A STA $DC0D      zulassen
037D LDA #$FE       Timer B
037F STA $DC06      von CIA 1
0382 LDA #$01       auf $01FE
0384 STA $DC07      setzen
0387 LDA #$19       Timer starten
0389 STA $DC0F
038C LDA $DC0D      Interrupt-Bits löschen
038F CLI            Interrupts wieder zulassen
0390 LDA $BF        Schieberegister auslesen
```

```
0392 CMP #$03       Drei gefunden?
0394 BNE $0390      verzweige, wenn nein
0396 LDA #$07       Bit-Zähler auf Sieben
0398 STA $B4        setzen
039A LDX #$10       Zähler für Synchronisationsmarkierung
039C STX $9C        Signal für 'Byte gelesen' rücksetzen
039E JSR $02CA      Byte lesen
03A1 CMP #$03       gleich Drei?
03A3 BNE $0390      verzweige, wenn nein
03A5 DEX            schon $10 mal das Byte Drei gefunden?
03A6 BNE $039E      verzweige, wenn nein
03A8 JSR $02CA      Byte lesen
03AB CMP #$03       gleich Drei?
03AD BEQ $03A8      verzweige, wenn ja
03AF CMP #$48       gleich $48?
03B1 BNE $0390      verzweige, wenn nein
03B3 JSR $02CA      Schleife: Anfangs-, Endadreß-Bytes lesen
03B6 STA $AC,X      nach $AC bis $AF bringen
03B8 INX            Zeiger erhöhen
03B9 CPX #$04       schon vier Bytes geholt?
03BB BNE $03B3      verzweige, wenn nein
03BD JSR $02CA      Programm-Byte lesen
03C0 STA ($AC),Y    in Speicher schreiben
03C2 EOR $9B        Prüfsumme bilden
03C4 STA $9B        und speichern
03C6 JSR $FCDB      $AC/$AD erhöhen
03C9 JSR $FCD1      schon Endadresse $AE/AF erreicht?
03CC BNE $03BD      verzweige, wenn nein
03CE JSR $02CA      Prüfsumme lesen
03D1 CMP $9B        mit gebildeter Prüfsumme vergleichen
03D3 BEQ $03D8      verzweige, wenn gleich
03D5 JMP $FCE2      sonst RESET
03D8 SEI            Interrupts verhindern
03D9 JSR $FD15      Interruptvektor zurücksetzen
03DC JSR $FDA3      CIAs wieder auf alten Interruptbetrieb
03DF JSR $FC93      Motor aus, Bildschirm an
03E2 LDA #$0E       Randfarbe des Bildschirms
03E4 STA $D020      wieder auf Standardwert
03E7 LDX $AE        LOW-Byte der Endadresse
03E9 LDY $AF        HIGH-Byte der Endadresse
```

```
03EB CLI          Interrupts zulassen
03EC JMP $02D9    Endadresse in BASIC-Programmendezeiger
03EF LDA #$E1     $E1 nach LOW-Byte des STOP-Vektors
03F1 STA $0328    und so STOP sperren
03F4 LDA #$00
03F6 JSR $A871    RUN-Befehl für Autostart
03F9 JMP $A7AE    zurück zur Interpreterschleife
```

Der nächste Teil der Laderoutine beginnt im Speicher bei $C23F, wird aber zum Ablauf nach $02A8 geladen.

Interruptroutine für Laden:

```
02A8 LDA $DC0D    Interruptregister lesen
02AB LDX #$19     Timer neu starten
02AD STX $DC0F
02B0 LSR          Bit für Timer B
02B1 LSR          ins Carry-Bit schieben
02B2 ROL $BF      als gelesenes Bit ins Schieberegister
02B4 INC $D020    optische Kontrolle
02B7 DEC $B4      schon komplettes Byte gelesen?
02B9 BPL $02C7    verzweige, wenn nein
02BB LDA #$07     Bit-Zähler auf Sieben
02BD STA $B4      setzen
02BF LDA $BF      Schieberegister nach
02C1 STA $BE      Zwischenspeicher für gelesenes Byte
02C3 LDA #$80     Signal 'Byte gelesen'
02C5 STA $9C      setzen
02C7 JMP $FEBC    Interrupt-Ende
```

gelesenes Byte holen:

```
02CA BIT $9C      Byte schon gelesen?
02CC BPL $02CA    verzweige, wenn nein
02CE STY $9C      'Byte gelesen'-Signal löschen
02D0 LDA $BE      Byte holen
02D2 RTS          Rücksprung
```

Einsprung für Autostart:

```
02D3 JSR $E453      BASIC-Vektoren rücksetzen
02D6 JMP $0351      Sprung zur Laderoutine
02D9 STX $2D        BASIC-Ende LOW-Byte setzen
02DB STY $2E        BASIC-Ende HIGH-Byte setzen
02DE JSR $A569      CLR
02E1 JSR $A533      Programmzeilen binden
02E4 JMP $03EF      unbedingter Sprung
```

Das Kassetten-Kopierschutz-System KKS (BASIC-Lader):

```
100 FORI=1TO636STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
:NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATAA9,0B,8D,32,03,A9,C0,8D,33,03,60,A5,BA,C9,01, 43
301 DATAF0,03,4C,ED,F5,A2,00,BD,3E,C2,9D,A8,02,E8,E0, 143
302 DATA48,D0,F5,A0,00,C4,B7,F0,0A,B1,BB,99,83,C1,C8, 51
303 DATAC0,10,D0,F2,C0,10,F0,09,A9,20,99,83,C1,C8,4C, 21
304 DATA31,C0,A9,01,AA,A8,20,BA,FF,A9,BC,A2,83,A0,C1, 177
305 DATA20,BD,FF,A5,AE,48,A5,AF,48,A5,C1,48,A5,C2,48, 112
306 DATAA9,A8,85,C1,A9,02,85,C2,A9,04,85,AE,A9,03,85, 154
307 DATAAF,A9,D3,8D,02,03,A9,02,8D,03,03,A9,00,85,9D, 198
308 DATA20,ED,F5,68,85,AD,EA,EA,85,C2,68,85,AC,EA,EA, 36
309 DATA85,C1,68,85,AF,68,85,AE,20,53,E4,78,AD,11,D0, 218
310 DATA29,EF,8D,11,D0,A9,44,8D,14,03,A9,C1,8D,15,03, 38
311 DATAA5,01,48,A9,06,85,01,85,C0,A9,03,A0,00,84,9B, 211
312 DATACA,D0,FD,88,D0,FA,38,E9,01,D0,F5,A9,7F,8D,0D, 146
313 DATADC,A9,82,8D,0D,DC,A9,47,8D,06,DC,E8,8E,07,DC, 53
314 DATAA9,07,85,B4,A9,19,8D,0F,DC,AD,0D,DC,58,A9,03, 189
315 DATA20,7A,C1,88,D0,F8,A9,48,20,7A,C1,A5,C1,20,7A, 247
316 DATAC1,A5,C2,20,7A,C1,A5,AE,38,E5,AC,48,A5,AF,E5, 32
317 DATAAD,AA,68,18,65,C1,20,7A,C1,8A,65,C2,20,7A,C1, 100
318 DATAB1,AC,AA,45,9B,85,9B,8A,20,7A,C1,20,DB,FC,20, 3
319 DATAD1,FC,D0,ED,A5,9B,20,7A,C1,20,7A,C1,20,7A,C1, 219
```

```
320 DATA78,68,09,20,85,01,20,15,FD,20,A3,FD,20,93,FC, 48
321 DATAA9,0E,8D,20,D0,58,4C,74,A4,06,BF,90,05,A2,33, 31
322 DATACA,D0,FD,A5,01,29,F7,85,01,A2,19,8E,0F,DC,A2, 185
323 DATA10,CA,D0,FD,09,08,85,01,EE,20,D0,C6,B4,10,0C, 178
324 DATAA9,07,85,B4,A5,BE,85,BF,A9,80,85,9C,AD,0D,DC, 112
325 DATA4C,BC,FE,24,9C,10,FC,85,BE,84,9C,60,20,20,20, 245
326 DATA20,20,20,20,20,20,20,20,20,20,20,20,78,A9, 193
327 DATA0B,8D,11,D0,A9,A8,8D,14,03,A9,02,8D,15,03,A5, 99
328 DATA01,29,1F,85,01,85,C0,A0,00,84,9B,CA,D0,FD,88, 242
329 DATAD0,FA,A9,7F,8D,0D,DC,A9,90,8D,0D,DC,A9,FE,8D, 75
330 DATA06,DC,A9,01,8D,07,DC,A9,19,8D,0F,DC,AD,0D,DC, 204
331 DATA58,A5,BF,C9,03,D0,FA,A9,07,85,B4,A2,10,86,9C, 15
332 DATA20,CA,02,C9,03,D0,EB,CA,D0,F6,20,CA,02,C9,03, 187
333 DATAF0,F9,C9,48,D0,DD,20,CA,02,95,AC,E8,E0,04,D0, 112
334 DATAF6,20,CA,02,91,AC,45,9B,85,9B,20,DB,FC,20,D1, 7
335 DATAFC,D0,EF,20,CA,02,C5,9B,F0,03,4C,E2,FC,78,20, 188
336 DATA15,FD,20,A3,FD,20,93,FC,A9,0E,8D,20,D0,A6,AE, 9
337 DATAA4,AF,58,4C,D9,02,A9,E1,8D,28,03,A9,00,20,71, 78
338 DATAA8,4C,AE,A7,AD,0D,DC,A2,19,8E,0F,DC,4A,4A,26, 205
339 DATABF,EE,20,D0,C6,B4,10,0C,A9,07,85,B4,A5,BF,85, 5
340 DATABE,A9,80,85,9C,4C,BC,FE,24,9C,10,FC,84,9C,A5, 159
341 DATABE,60,20,53,E4,4C,51,03,86,2D,84,2E,20,59,A6, 153
342 DATA20,33,A5,4C,EF,03,A4,A4,A4,A4,A4,A4,A4,A4,A4, 250
```

5.5.4 Wichtige Betriebssystemadressen und -routinen

Im KKS werden einige Adressen und Unterroutinen des Betriebssystems verwendet. Diese und noch einige andere können beim Erstellen von Kopierschutzverfahren und ähnlichen Programmen für die Datasette äußerst hilfreich sein. Aus diesem Grunde sind die wichtigsten im folgenden aufgelistet:

Zeropage-Adressen:

```
Hexadez. Dezimal Funktion
$2B/$2C   43/ 44 Anfang des BASIC-Speichers
$2D/$2E   45/ 46 Ende des BASIC-Speichers
$90       144    Status (entspricht Variable ST in BASIC)
```

$93	147	Flag für LOAD ($00) oder VERIFY ($01)
$9D	157	Flag für Direktmodus $80 oder Programm $00 ($C0 läßt Ausgabe aller Fehlermeldungen, also auch die des Betriebssystems, zu; $80 nur die des BASIC-Interpreters; $00 überhaupt keine)
$A6	166	Anzahl der schon aus dem Bandpuffer gelesenen oder in den Puffer hineingeschriebenen Bytes
$AC/$AD	172/173	laufende Adresse für LOAD/SAVE
$AE/$AF	174/175	Zeiger auf Endadresse + 1 für LOAD/SAVE
$B2/$B3	178/179	Zeiger auf Beginn des Bandpuffers
$B7	183	Länge des Filenamens
$B8	184	logische Filenummer
$B9	185	Sekundäradresse
$BA	186	Geräteadresse
$BB/$BC	187/188	Zeiger auf Filenamen
$C0	192	Wenn bei gedrückter Datasetten-Taste der Inhalt dieser Speicherstelle ungleich null ist, so wird der Motor nicht eingeschaltet
$C1/$C2	193/194	Startadresse für Ein-/Ausgabe
$C3/$C4	195/196	Endadresse für Ein-/Ausgabe

Betriebssystem- und BASIC-Interpreter-Routinen:

$A474 Einsprung in Eingabeschleife des BASIC-Interpreters
$A533 Programmzeilen neu binden
$A659 CLR-Befehl
$A7AE Befehlsausführungsschleife des BASIC-Interpreters
$A871 RUN-Befehl
$E17A Ladefehler testen und ggf. anzeigen, dann wie $E1A1
$E1D4 Filename, Sekundär-, Geräteadr. aus BASIC-Text holen
$E453 Zurücksetzen aller BASIC-Interpreter-Vektoren
$F68F 'SAVING "Filename"' ausgeben
$F750 Ausgabe 'FOUND "Filename"', wartet auf 'C='
$F80D Bandpufferzeiger $A6 erhöhen
$F817 Wartet auf Bandtaste, Ausgabe "PRESS PLAY ON TAPE"
$F838 Wartet auf Bandtaste, "PRESS RECORD AND PLAY ON TAPE"
$FC93 Rekorderbetrieb beenden

$FCD1 Vergleich von $AE/$AF mit $AC/AD
$FCDB $AC/$AD inkrementieren
$FCE2 RESET, Rechner in Einschaltzustand versetzen
$FDA3 CIAs initialisiern
$FEBC Abschluß der IRQ-Routine
$FF8A Zurücksetzen aller Betriebssystem-Vektoren
$FFBA FILPAR, setzt Geräte-, Sekundäradresse, Filenummer
$FFBD FILNAM, setzt Filenamen-Parameter
$FFC0 OPEN
$FFC3 CLOSE
$FFC6 CHKIN, Eingabegerät setzen
$FFC9 CHKOUT, Ausgabegerät setzen
$FFCC CLRCH, Ein-, Ausgabe zurücksetzen
$FFCF BASIN, Eingabe eines Zeichens
$FFD2 BASOUT, Ausgabe eines Zeichens
$FFD5 LOAD (bzw. VERIFY)
$FFD8 SAVE
$FFE7 CLALL, alle offenen Kanäle schließen

6. Diskettenkopierschutz

Wie arbeitet ein solcher Diskettenkopierschutz? Wie und wofür kann ich ihn selber benutzen? Das sind die Fragen, die in dem folgenden Kapitel beantwortet werden sollen.

Wofür Sie einen Kopierschutz brauchen, ist einfach zu beantworten. Sie können mit Hilfe eines solchen Schutzes sicherstellen, daß Ihr mühsam geschriebenes Programm nicht ohne Ihr Wissen weitergegeben werden kann. Diese in diesem Buch erläuterten Schutzverfahren sind auch ohne weiteres dazu geeignet, professionelle Software zu schützen. Sie entsprechen nicht den in einigen Zeitschriften oder ähnlichem beschriebenen Verfahren, die mit jedem drittklassigen Kopierprogramm kopiert werden können.

Selbst wenn Sie kein Interesse haben, Ihre Software zu schützen, ist dieses Kapitel auch im Hinblick auf die interessante Floppy-Programmierung sehr zu empfehlen.

Auf die Frage, wie ein solcher Kopierschutz arbeitet, ist allgemein zu sagen, daß es sich hierbei um eine Manipulation auf der geschützten Diskette handelt, die nicht ohne weiteres von einem Kopierprogramm kopiert werden kann. Wie dieses im einzelnen aussieht, wird im Laufe dieses Buches noch besprochen.

Leider müssen wir zum Verständnis der folgenden Programme ein recht gutes Wissen über das Arbeiten der Floppy sowie Kenntnisse der Programmierung in Maschinensprache voraussetzen, da derart weitreichende Einführungen den Rahmen dieses Buches sprengen würden. Für diejenigen, die diese Kenntnisse nicht haben, ist dieses Kapitel dennoch interessant, weil alle Schutzsysteme auch in Form eines BASIC-Loaders vorliegen und somit für jeden nutzbar sind.

6.1 Unser "Handwerkszeug"

Bevor wir uns intensiver mit der Programmierung eines Kopierschutzes befassen, wollen wir zuvor näher auf das "In-

nenleben" der Floppy eingehen, um Ihnen das Verständnis der zu besprechenden Kopierschutzsysteme zu ermöglichen. Wenn Sie über dieses Wissen bereits verfügen, können Sie sich sofort dem Auftragen eines Kopierschutzes widmen und die Kapitel bis 6.2 überspringen.

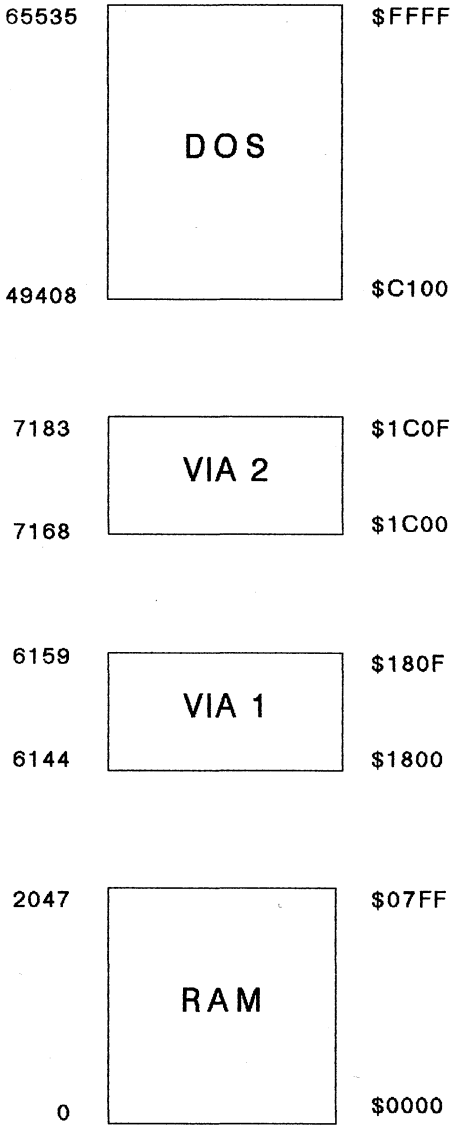
An dieser Stelle nun die Übersicht über die Speicheraufteilung der Floppy und eine Übersicht über die Nutzung der Puffer der Floppy.

VIA 1 ist der Baustein, der den Datenaustausch zwichen Computer und Rechner herstellt. Er verwaltet den Serielle-Bus.

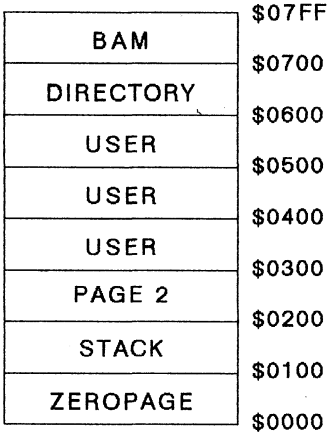
VIA 2 wird ausschließlich für den Datenaustausch zwischen Floppy und Diskette verwendet.

Es ist noch wissenswert, daß das RAM der Floppy ab $8000 gespiegelt ist, was bedeutet, das die Speicherstellen ab $8000 die gleichen Werte und auch die gleiche Bedeutung wie die Speichestellen ab $0000 haben.

SPEICHERAUFTEILUNG



RAMBELEGUNG



6.1.1 Die Arbeitsweise des DOS

Die VC1541 ist ein intelligentes Diskettenlaufwerk mit eigenem Mikroprozessor und Betriebssystem (Disk Operating System, DOS). Dadurch wird kein Speicherplatz und keine Rechenzeit des angeschlossenen Rechners benötigt. Der Rechner braucht der Floppy lediglich Befehle zu übermitteln, die diese dann selbsttätig ausführt.

Die Floppy hat damit drei Aufgaben gleichzeitig zu erledigen: Zum einen muß sie den Datenverkehr vom und zum Rechner durchführen. Die zweite Aufgabe ist die Interpretation der Befehle, die Verwaltung von Dateien, der zugeordneten Übertragungskanäle und der Blockpuffer. Die dritte Aufgabe ist die hardwaremäßige Bedienung der Diskette; dazu gehört das Schreiben und Lesen einzelner Blocks auf der Diskette sowie das Formatieren von Disketten.

Diese Aufgaben muß bei der VC1541 ein 6502-Mikroprozessor gleichzeitig durchführen. Dies ist nur mit Hilfe der Interrupt-Technik möglich. Nur so können drei Programme quasi gleichzeitig ablaufen.

Nachdem wir schon etwas näher auf den Aufbau der Diskette eingegangen sind, wollen wir uns einmal ansehen, wie das DOS alle seine Aufgaben realisiert. Dies ist besonders dann wichtig, wenn man selbst Programme in der Floppy schreiben will, wie beispielsweise seinen eigenen Kopierschutz.

Beim Schreiben von Programmen im Floppypuffer kann man selbstverständlich auch viel vom Betriebssystem der Floppy Gebrauch machen, das für die meisten Aufgaben schon Routinen parat hat. Doch in der Floppy ist dies nicht so problemlos wie im C64. Das Betriebssystem der Floppy kann man in zwei Teile unterteilen. Der erste Teil ist das Hauptprogramm, das in einer Endlosschleife läuft. Von diesem Programm aus wird hauptsächlich der serielle Bus verwaltet. Fast alle Unterprogrammaufrufe werden mit absoluten Sprüngen realisiert und müssen somit auch mit einem JMP-Befehl zurück ins Hauptprogramm abgeschlossen werden. Diese Routinen können deswegen

kaum für die eigene Arbeit verwendet werden, da sie nach einmaligem Aufruf sofort ins Hauptprogramm springen. Sie müssen solche Routinen also selbst schreiben und können nicht auf die vorhandenen Routinen des Betriebsystems zurückgreifen.

Der zweite Teil des Betriebsystems ist dafür um so besser für eigene Programme zu verwenden. Es handelt sich hierbei um ein Interrupt-Programm, die sogenannte 'Jobschleife'.

Dieses Programm übermimmt die Lese- und Schreiboperationen auf und von Diskette. Bei jedem Interrupt werden die Speicherstellen $00 bis $05 der Zeropage, die die 'Schnittstelle' zwischen dem Hauptprogramm und Interruptprogramm herstellen, auf ihre Werte überprüft. Werte, die größer oder gleich $80 (128) sind, werden als Befehle (Jobs) erkannt und daraufhin ausgeführt. Jede dieser Speicherstellen (Job-Speicher) bezieht sich auf einen bestimmten Speicherbereich. Zusätzlich gehören zu jedem Job-Speicher noch zwei Speicherstellen, die den Track und den Sektor angeben, auf die sich der Job (Befehl) bezieht.

Nachfolgend eine Tabelle, welche den Job-Speichern ihre Track und Sektorangabe sowie ihren Speicherbereich zuordnet:

JOB	TRACK	SEKTOR	SPEICHERBEREICH
$00	$06	$07	$0300-$03FF
$01	$08	$09	$0400-$04FF
$02	$0A	$0B	$0500-$05FF
$03	$0C	$0D	$0600-$06FF
$04	$0E	$0F	$0700-$07FF
$05	$10	$11	kein RAM

Anschließend folgt eine Tabelle, die die Job-Codes und ihre Bedeutung zeigt.

JOB-CODE	BEDEUTUNG
$80 128	Sektor lesen
$90 144	Sektor schreiben
$A0 160	Sektor verifizieren
$B0 176	Sektor suchen
$C0 192	Bump, Kopfanschlagen
$D0 208	Programm im Puffer ausführen
$E0 224	Programm im Puffer ausführen, vorher Laufwerksmotor einschalten und Kopf positionieren

Machen wir uns die Verarbeitung der Job-Codes anhand eines Beispiels besser verständlich:

Wenn vom BASIC aus durch einen 'U1' Befehl ein Block gelesen werden soll, so nimmt das Hauptprogramm diesen Befehl entgegen, erkennt ihn und veranlaßt, daß der entsprechende Block gelesen wird. Diese Veranlassung sieht jetzt so aus, daß die Track und Sektornummer in die Speicherspellen, die im Zusammenhang mit dem angegebenen Puffer stehen, geschrieben werden. Daraufhin wird der Job-Code $80 (Sektor lesen) in den entsprechenden Job-Speicher geschrieben.

Nehmen wir an, zum Lesen dieses Blocks wäre Puffer 2 reserviert worden; dann wäre die Track- und Sektornummer in die Speicherstellen $0A und $0B geschrieben worden. Der Job-Code $80 wäre danach in den Job-Speicher $02 geschrieben worden. Nach diesen Arbeitsgängen beginnt die Aufgabe der Jobschleife. Diese überprüft jetzt die Job-Speicher, findet im Job-Speicher die $80 und holt sich aus $0A und $0B die Track- und Sektornummer des zu lesenden Sektors. Daraufhin wird der Kopf positioniert und der Sektor gelesen. Die gelesenen Daten werden im Puffer 2 ($0500-$05FF) abgelegt.

Das Hauptprogramm befindet sich während dieser Zeit in einer Wartestellung. Es wartet auf die READY-Meldung des Jobs. Diese Rückmeldung wird in den selben Job-Speicher geschrieben, in den auch der Befehl geschrieben wurde. Die Rückmel-

dungen unterscheiden sich von den Job-Codes dadurch, daß bei den Job-Codes das höchstwertige Bit gesetzt und bei den Rückmeldungen gelöscht ist. Das Hauptprogramm überprüft immer das höchstwertigste Bit und beginnt, sobald die Rückmeldung kommt, diese auszuwerten.

Bevor wir auf die Rückmeldungen eingehen, noch ein Hinweis zu den Job-Codes, die ein Programm im Puffer ausführen. Der Job-Code $D0 startet ein Programm bei der Anfangsadresse des jeweiligen Puffers. Der Befehl $E0 fährt den Laufwerkmotor zuvor noch hoch, positioniert den Kopf und führt dann wie $D0 das Programm im angegebenen Puffer aus.

Wollen wir also den Inhalt von Track 24, Sektor 8 in den Puffer #0 ($300 - $3FF) lesen, kann dies durch folgende Routine geschehen:

```
100 OPEN 1,8,15
110 TRACK = 24: SECTOR = 8: LESEN = 128
120 PRINT#1, "M-W"CHR$(6)CHR$(0)CHR$(2)CHR$(TRACK)CHR$(SECTOR)
130 PRINT#1, "M-W"CHR$(0)CHR$(0)CHR$(1)CHR$(LESEN)
140 PRINT#1, "M-R"CHR$(0)CHR$(0)
150 GET#1, A$:IF A$="" THEN A$=CHR$(0)
160 IF ASC(A$)>127 THEN 140
170 PRINT "FEHLERCODE ="; ASC(A$)
180 CLOSE 1
```

In Zeile 120 werden Track- und Sektornummer für Puffer #0 übergeben, in Zeile 130 der Befehl zum Lesen. Die Programmschleife in Zeile 140 bis 160 wartet das Ende des Jobs ab. In Zeile 170 wird der Fehlercode ausgeben. Diese Codes haben folgende Bedeutung:

Code	BEDEUTUNG	DOS-Fehlermeldung
$01	alles ok	00, OK
$02	Headerblock nicht gefunden	20, READ ERROR
$03	SYNC nicht gefunden	21, READ ERROR
$04	Datenblock nicht gefunden	22, READ ERROR
$05	Checksummenfehler im Datenblock	23, READ ERROR
$07	Verify-Fehler	25, WRITE ERROR
$08	Diskette schreibgeschützt	26, WRITE PROTECT ON
$09	Checksummenfehler im Headerblock	27, READ ERROR
$0B	Falsche ID gelesen	29, ID MISMATCH
$0F	Diskette nicht eingelegt	74, DRIVE NOT READY

Die im Floppyhandbuch aufgeführten Fehlermeldungen '24, READ ERROR' (GCR-Kodierung wird nicht erkannt) und '28, WRITE ERROR' treten bei der VC1541 nicht auf. Zur Bedeutung der Fehlermeldungen sehen Sie bitte im Anhang nach.

Zur Benutzung der Disk-Controller-Routinen sei noch folgendes vermerkt: Wenn Sie einen Block mit dem 'U1'-Befehl lesen, so wird der Befehl 'Lese Block' an die Disk-Controller übergeben. Wird von diesem nun ein Fehler gemeldet, so veranlassen die 'logischen DOS-Routinen' (das Hauptprogramm) weitere Leseversuche jeweils eine halbe Spur links und rechts vom Track. Hat auch dies keinen Erfolg, wird ein 'Bump' durchgeführt (der Kopf geht nach Track 1 und gibt die typischen Geräusche bei einem Lesefehler von sich) und das ganze wird nochmal versucht. Normalerweise werden vom DOS 5 Leseversuche gemacht, ehe endgültig ein Fehler gemeldet wird. Diesen Mechanismus umgehen wir, wenn wir direkt die Disk-Controller-Routinen ansprechen. Für normale Schreib- und Leseoperationen sollten daher weiterhin die 'U1' und 'U2'-Befehle benutzt werden.

6.1.2 Das Disketten-Aufzeichnungsverfahren

In diesem Kapitel versuchen wir, Ihnen näher zu bringen, wie die Bits auf die Diskette kommen und von dort wieder gelesen werden können. Wie Sie bereits wissen, ist die Diskette in 35 Spuren oder Tracks unterteilt, die als konzentrische Ringe auf der Diskette angeordnet sind. Die äußerste Spur erhält die Nummer 1, und Track 35 ist die innerste Spur. Um die einzelnen Spuren anzusteuern, hat das Laufwerk einen Schritt- oder Steppermotor, mit dem der Schreib/Lesekopf über jeden Track positioniert werden kann. Die Aufzeichnung der Daten geschieht nun bitweise. Dabei wird jedes '1'-Bit durch einen Wechsel der Magnetisierungsrichtung gekennzeichnet, während bei einem '0'-Bit nichts passiert. Die Bits werden in einem festen Takt geschrieben, der ca. 250000 Bits/s beträgt.

Wenn wir nun Daten nach diesem Verfahren auf eine Spur schreiben, sind wir anschließend nicht mehr in der Lage, die Daten korrekt wieder zu lesen. Da sich die Diskette fortlaufend dreht, die Spuren also keinen Anfang und kein Ende haben, können wir den Beginn unserer Aufzeichnung nicht ermitteln. Doch selbst wenn wir den Anfang der Spur feststellen könnten, würden wir Probleme haben, den Beginn jedes Bytes festzustellen. Theoretisch ist durch den Beginn der Aufzeichnung auch der Beginn jedes einzelnen nachfolgenden Bytes bestimmt. Dies setzt jedoch voraus, daß die Umdrehungsgeschwindigkeit der Diskette immer absolut gleichbleibend ist, auch von einem Drive zum anderen. Diese Forderung ist jedoch illusorisch, und man hat sich eine andere Möglichkeit überlegt, wie man den Beginn der einzelnen Bytes sicher erkennen kann.

Zuerst unterteilt man einen Track in mehrere Sektoren. Da auf einer Spur bei der oben genannten Bitrate mehr als 6000 Bytes untergebracht werden können, braucht man zum einen einen Pufferspeicher dieser Größe in der Floppy. Ein weiterer Nachteil wäre es, daß nur maximal 35 Dateien abzuspeichern wären. Wenn wir ein Programm von 1 KByte Länge abspeichern würden, so wären 5 KByte verschenkt, die Ausnutzung der Diskette also sehr unökonomisch. Da die Tracks von außen nach innen schmaler werden, werden die äußeren Tracks in mehr Sektoren

unterteilt als die inneren. Dazu muß die Bitrate nach außen hin gesteigert werden. Die folgende Tabelle enthält die entsprechenden Daten.

TRACK	BITRATE	BYTES PRO TRACK
1-17	307692	7692
18-24	285714	7143
25-30	266667	6667
31-35	250000	6250

Doch damit hat sich unserer Problem nur vergrößert: Wie erkennen wir den Beginn der Sektoren auf dem Track? Wir müssen also bestimmte Bitkombinationen bevorzugt erkennen können, die als Daten-Bytes nicht vorkommen können. Mit acht Bit lassen sich 256 Kombinationen darstellen, die es auch alle als Daten-Bytes kann. Der Schlüssel zur Lösung liegt darin, ein Byte nicht durch 8, sondern für die Diskettenaufzeichnung durch mehr Bits darzustellen. Damit sind wir schon beim Group Code Recording, wie es von Commodore verwendet wird.

Wir benutzen als sogenanntes Synchron- oder SYNC-Zeichen die Bitkombination %11111111 oder $FF. Die Daten-Bytes müssen nun so verschlüsselt werden, daß niemals 8 '1'-Bits hintereinander vorkommen. Das Group Code Recording, kurz GCR genannt, bildet nun jeweils 4 Bits auf 5 Bits ab. Die 5-Bit-Kombination ist so gewählt, daß die obige Forderung erfüllt ist. Eine zusätzliche Forderung besteht noch darin, dafür zu sorgen, daß auch niemals mehr als zwei '0'-Bit hintereinander folgen, da dabei ja kein Wechsel der Magnetisierungsrichtung erfolgt und es dadurch Probleme beim Lesen der Daten geben kann. Die Tabelle gibt nun an, wie jeweils 4 Bits verschlüsselt werden.

ORIGINALDATEN		GCR-CODE
$0	0000	01010
$1	0001	01011
$2	0010	10010
$3	0011	10011
$4	0100	01110
$5	0101	01111
$6	0110	10110
$7	0111	10111
$8	1000	01001
$9	1001	11001
$A	1010	11010
$B	1011	11011
$C	1100	01101
$D	1101	11101
$E	1110	11110
$F	1111	10101

Durch beliebiges Aneinandereihen des GCR-Codes kann also niemals ein Bitmuster entstehen, das mehr als 8 aufeinanderfolgende '1'-Bits oder mehr als zwei '0'-Bits enthält. Wir können also jetzt den Beginn eines Sektors durch SYNC-Bytes markieren, die durch eine Digitalschaltung erkannt und gemeldet werden. Doch wie wissen wir, welcher Sektor nach einem SYNC-Byte folgt? Dazu steht vor jedem Sektor ein sogenanntes Blockheaderfeld (Sektorkopf), das Track- und Sektornummer sowie zusätzlich noch die Identifikation der Diskette enthält (das sind die zwei Zeichen, die Sie beim Formatieren der Diskette mit angeben). Der nachfolgende eigentliche Sektorinhalt wird wieder durch vorausgehende SYNC-Bytes eingeleitet. Damit man Blockheader und Datenblock unterscheiden kann, folgt nach dem SYNC-Zeichen ein Kennbyte. Ein Headerblock wird durch eine 8 identifiziert, ein Datenblock durch eine 7. Damit Lesefehler, die z.B. durch eine verschmutzte oder zerkratzte Diskette entstehen können, erkannt werden, enthält jeder Block noch eine Checksumme über alle Bytes. Ein Blockheader hat damit folgenden Aufbau:

```
  5   SYNC-ZEICHEN
  1   HEADERBLOCKKENNZEICHEN ($08)
  1   CHECKSUMME ÜBER ALLE BYTES
  1   SEKTOR-NUMMER
  1   TRACK-NUMMER
  1   ID2
  1   ID1
  2   ABSCHLUSSBYTES ($0F)
  8   FÜLLBYTES ($55)
```

Bis auf die SYNC-Zeichen ($FF, %11111111) und die 8 Füllbytes ($55 %01010101) werden alle anderen Bytes in GCR-Code umgewandelt. Die Füllbytes sind erforderlich, da immer nur eine durch 4 teilbare Anzahl Bytes in GCR-Code umgewandelt werden kann. Aus einem Byte (8 Bits) werden nach der Umwandlung 10 Bits. 4 Bytes (32 Bits) werden in 40 Bits umgewandelt, was gerade 5 Bytes entspricht. Das folgende Schema soll deutlich machen, wie die Umwandlung vonstatten geht.

```
ORIGINALDATEN   11112222 33334444 55556666 77778888
GCR-DATEN       11111222 22333334 44445555 56666677 7778888
```

Eine Zifferngruppe soll dabei ein Nibble (4 Bits) darstellen. Die Umwandlung geschieht von einer DOS-Routine mit Hilfe der GCR-Code-Tabelle, die ab Adresse $F77F im ROM steht.

Doch jetzt zum Aufbau des Datenblocks.

```
      5   SYNC-ZEICHEN
      1   DATENBLOCKKENNZEICHEN ($07)
    256   DATEN-BYTES
      1   CHECKSUMME
      2   ABSCHLUSSBYTES ($00)
MIN.  4   FÜLLBYTES ($55)
```

Der Block wird wieder mit 5 SYNC-Zeichen eingeleitet, dann folgen im GCR-Code das Kennzeichen für den Datenblock, eine 7, dann die 256 Daten-Bytes. Zur Fehlererkennung folgt eine Ein-Byte-Checksumme (Exklusiv-Oder-Verknüpfung aller Bytes) und zwei Nullbytes als Abschlußbytes. Die Anzahl der Füllbytes bis zum Beginn des nächsten Headerblocks ist variabel und wird bei der Formatierung aus der Gesamtkapazität des jeweiligen Tracks ermittelt.

Ein kompletter Sektor auf Diskette enhält also 8 (Headerblock) + 260 (Datenblock) Bytes, die in GCR-Code gewandelt 335 Bytes ergeben, sowie 10 SYNC-Zeichen und mindestens 12 Füllbytes. Damit besteht ein Sektor aus mindestens 357 Bytes im Verhältnis zu 256 Nutzbytes. Es gehen also ca. 28% der gesamten Diskettenkapazität für die Datenorganisation verloren.

Das Beschreiben der Diskette geschieht bereits komplett beim Formatieren. Sehen wir uns jetzt einmal an, wie die Floppy einen Sektor liest.

1. Laufwerksmotor einschalten.
2. Schreib/Lesekopf über Track bringen.
3. Headerblock für gewünschten Track und Sektor erzeugen und in GCR-Code umwandeln.
4. SYNC-Zeichen abwarten.
5. Die nächsten 8 Bytes lesen und mit dem oben erzeugten GCR-Code vergleichen.
6. Falls keine Übereinstimmung: zurück zu Schritt 4.
7. Der richtige Sektorheader wurde gefunden, jetzt SYNC-Zeichen des Datenblocks abwarten.
8. Datenblock lesen und GCR-Code decodieren.

Dies sind, vereinfacht dargestellt, die Schritte, die beim Lesen eines Sektors durchlaufen werden. Nicht berücksichtigt wurde dabei, was im Fehlerfall passiert, z.B. falls ein Checksummenfehler festgestellt wurde oder falls schon der Headerblock nicht korrekt gelesen werden konnte.

Das Schreiben eines Blocks verläuft ähnlich. Auch hier muß zuerst der korrekte Headerblock gefunden werden, ehe man den alten Datenblock mit den neuen Daten überschreiben kann.

1. Laufwerksmotor einschalten.
2. Schreib/Lesekopf über Track bringen.
3. Datenblock in GCR-Code umwandeln.
4. Headerblock für gewünschten Track und Sektor erzeugen und in GCR-Code umwandeln.
5. SYNC-Zeichen abwarten.
6. Die nächsten 8 Bytes lesen und mit dem oben erzeugten GCR-Code vergleichen.
7. Falls keine Übereinstimmung zurück zu Schritt 5.
8. 8 GAP-Bytes überlesen.
9. 5 SYNC-Zeichen schreiben.
10. GCR-Daten schreiben.
11. Geschriebenen Sektor lesen und mit den Daten im RAM vergleichen (Verify).

Beim Schreiben eines Blocks wird also nur der Datenblock mit den zugehörigen SYNC-Bytes geschrieben. Der Blockheader bleibt unverändert, er wird nur einmal bei der Formatierung erzeugt.

6.1.3 Einführung in die Lese- und Schreibtechnik

Bevor wir dazu übergehen, die einzelnen Kopierschutzverfahren zu erläutern, sollte erst grundlegend die Methodik des Lesens und Speicherns von Daten auf Diskette besprochen werden.

Die zur Floppy-Steuerung interessantesten Register sind die Register 1 und 2 des VIA 2. Es sind die Adressen $1C00 und $1C01, wobei $1C00 hauptsächlich zur Ansteuerung der Laufwerk-LED und der Motoren und $1C01 zum Lesen und Schreiben der Daten benötigt werden.

Erläuterung der Bitfunktionen der Adresse $1C00

```
Bit

0 Steppermotorsteuerung
1 Steppermotorsteuerung
2 Laufwerksmotor;                    Bit=0: Motor aus
3 Laufwerk-LED;                      Bit=0: LED aus
4 Schreibschutz;                     Bit=0: Lichtschranke unterbrochen
5 Zur Einstellung der Bitrate zum
  Tonkopf (Speedflag)
6 Bedeutung wie Bit 5
7 SYNC-Signal beim Lesen gefunden; Bit=0: SYNC gefunden
```

Zum Lesen und Schreiben von Daten wird, wie schon gesagt, $1C01 benutzt. Schreibt man einen Wert in dieses Register, so wird dieser, sofern zuvor auf Schreiben umgestellt wurde, dieses Byte auf die Diskette geschrieben. Doch wie kann der Programmierer feststellen, wann das Byte vollständig geschrieben oder gelesen wurde? Um dies feststellen zu können, existiert eine BYTE-READY-Leitung, die mit dem OVERFLOW-Flag des Prozessorstatusregisters verbunden ist. Sobald ein Byte vollständig übertragen wurde, wird aufgrund dieser Leitung das OVERFLOW-Flag gesetzt. Dieses Flag wird von dieser Leitung zwar gesetzt, jedoch nicht wieder gelöscht. Dies muß der Programmierer mit Hilfe des CLV-Befehls 'von Hand' erledigen, damit er nicht ständig die Ready-Meldung für ein verarbeitetes Byte erhält. Die Zeit, in der ein Byte von Diskette gelesen oder auf Diskette geschrieben wird, liegt zwischen 26 und 32 Mikrosekunden, je nach Einstellung der Speed-Flags.

Zum Starten von Programmen in der Floppy gibt es grundsätzlich zwei Möglichkeiten. Zum einen können Sie Ihre Programme über die Jobcodes $E0 und $D0 als Interrupt-Programm starten, zum anderen ist es auch möglich, die Programme direkt zu starten.

Wir werden uns erst das Starten von Programmen mit Hilfe der Jobcodes ansehen.

Beispiel für das Laden eines Bytes:

```
0500 BVC $0500
0502 CLV
0503 LDA $1C01
```

In der ersten Zeile wird so lange gewartet, bis das V-Flag (BYTE-READY) auf HIGH gesetzt wird als Zeichen, daß ein Byte anliegt. Das V-Flag wird daraufhin 'von Hand' wieder auf LOW gesetzt. Anschließend wird das geladene Byte in den Akku geholt.

Beispiel für das Speichern eines Bytes:

```
0500 STA $1C01
0503 BVC $0503
0505 CLV
```

Das Speichern eines Bytes läuft, wie Sie sehen, analog dazu ab. Das Byte wird gespeichert, worauf auf BYTE-READY gewartet wird. Abschließend wird das V-Flag wieder gelöscht.

Vor dem Speichern muß man der Floppy noch mitteilen, daß die Daten, die in die Adresse $1C01 geschrieben werden, auch wirklich gespeichert werden. Dies erledigt das folgende kleine Programm.

```
LDA $1C0C
AND #$1F    PCR auf Schreibbetrieb umschalten
ORA #$C0
STA $1C0C
LDA #$FF    Port für Schreib-/Lesekopf
STA $1C03   auf Ausgang
```

Natürlich gibt es auch eine Routine zum Umschalten auf Lesen:

```
LDA $1C0C
ORA #$E0    PCR auf Lesebetrieb umschalten
STA $1C0C
LDA #$00    Port für Schreib-/Lesekopf
STA $1C03   auf Eingang
```

Diese Routine existiert auch in dieser Form im Betriebssystem der Floppy und läßt sich aufrufen mit:

```
JSR $FE00
```

Um Daten gezielt auf Diskette schreiben und auch wieder von dort lesen zu können, wird es häufig nötig sein, auf eine SYNC-Markierung zu warten. Auch für diesen Fall stellt das Betriebssystem der Floppy eine Routine zur Verfügung. Sie läßt sich aufrufen mit:

```
JSR $F556
```

Die Routine wartet 20 ms auf ein SYNC-Signal. Wird dieses nicht gefunden, wird eine entsprechende Fehlermeldung ausgegeben. Hier liegt auch der Nachteil im Benutzen der Routine. Der Programmierer hat keine Möglichkeit, selber auf ein eventuelles Nichtauffinden der SYNC-Markierung zu reagieren, da kein Rücksprung auf den Unterprogrammaufruf folgt.

Die Lösung ist einfach. Sie übernehmen die Routine in Ihr Programm und können somit bei negativer Abfrage zu Ihrer eigenen Routine verzweigen. Wenn Sie sicher sind, daß eine SYNC-Markierung gefunden wird, können Sie das entsprechende Bit auch ohne Fehlerabfrage direkt abfragen.

```
0500 BIT $1C00  Port abfragen
0502 BMI $0500  verzweige, wenn SYNC nicht gefunden
```

Das folgende Programm arbeitet selber mit Jobcodes und wird daher mit einem M-E-Befehl aufgerufen. Es erzeugt einen 22 READ-ERROR auf dem eingestellten Track und Sektor.

```
0600 LDA #$01    Nummer des zu ladenden Tracks
0602 STA $0A     in Trackspeicher für Puffer 2 schreiben
0604 LDA #$00    Nummer des zu ladenden Sektors
0606 STA $0B     in Sektorspeicher für Puffer 2 schreiben
0608 LDA #$80    Jobcode für Track/Sektor lesen
060A STA $02     in Jobspeicher des Puffer 2 schreiben
060C LDA $02     Jobspeicher lesen
060E BMI $060C   warten, bis Job abgearbeitet ist
0610 LDA #$09    eigenes Erkennungszeichen für Datenblock
0612 STA $47     in die entsprechende Adresse schreiben
0614 LDA #$90    Jobcode für Block speichern
0616 STA $02     in Jobspeicher für Puffer 2 schreiben
0618 LDA $02     Jobspeicher lesen
061A BMI $0618   warten, bis Job abgearbeitet ist
061C LDA #$07    normales Datenblock-Erkennungszeichen laden
061E STA $47     und speichern (rücksetzen)
0620 RTS         Rücksprung
```

In unserem Fall wird die Erkennungsziffer 7 durch die Ziffer 9 ersetzt. Der Daten-Header kann somit nicht mehr gefunden werden. Bei diesem ERROR handelt es sich um einen SOFT-ERROR, da die Daten des zerstörten Blocks sich trotz einer Fehlermeldung im Speicher der Floppy befinden. Vom BASIC oder von einem gewöhnlichen Diskmonitor aus ist es nicht möglich, diesen Block, ohne Umstellung der Speicherzelle $47 (71) auf 9, zu lesen.

Falls Sie diesen Block wieder reparieren wollen, reicht es aus, 0610 LDA #$09 durch 0610 LDA #$07 zu ersetzen und das Programm erneut zu starten. Sie speichern somit die Orginalkennzahl 7 mitsamt dem Block wieder ab.

Als nächstes stellen wir eine Routine vor, die den Tonkopf um beliebig viele Halbspuren nach innen oder außen bewegt. In diesem Beispiel wird der Tonkopf 16 Spuren nach außen und danach wieder nach innen gefahren. Es ist ratsam, die Diskette vor dem Ausprobieren dieser Routine zu initialisieren, damit der

Tonkopf nicht anstoßen kann. Für die Beobachtung des Tonkopfes bei diesem Beispiel und bei eigenen Experimenten ist es von Vorteil, die vier Schrauben an der Unterseite der Floppy zu lösen und den Deckel vorsichtig abzunehmem. Unserer Meinung nach überwiegen die Vorteile beim Arbeiten mit einer geöffneten Floppy im Gegensatz zu einer geschlossenen. Jetzt aber endlich das erwähnte Programm.

```
0500 SEI          Interrupt sperren
0501 LDA $1C00    Control-Register laden
0504 ORA #$04     Bit für Laufwerksmotor setzen
0506 STA $1C00    und wieder schreiben
0509 LDX #$30     Warteschleife
050B STX $4B      um dem
050D DEC $4B      Motor Zeit
050F BNE $050D    zum Hochfahren
0511 DEX          zu geben
0512 BNE $150D    verzweige, wenn Zeit nicht abgelaufen
0514 LDY #$10     Zähler für Stepanzahl auf $10 setzen
0516 JSR $0530    Kopf um eine Halbspur nach außen schieben
0519 DEY          Zähler verringern
051A BNE $0516    verzweige, wenn noch nicht abgezählt
051C LDY #$10     Zähler erneut setzen
051E JSR $0537    Kopf um eine Halbspur nach innen schieben
0521 DEY          Zähler verringern
0522 BNE $051E    verzweige wenn noch nicht abgezählt
0524 LDA $1C00    Control-Register laden
0527 AND #$F8     Bits 0 bis 2 löschen, um Motor zu stoppen
0529 STA $1C00    und wieder schreiben
052C CLI          Interrupt wieder ermöglichen
052D RTS          Rücksprung
052E NOP
052F NOP
```

Routine zur Kopfsteuerung

```
0530 LDX $1C00  Control-Register laden
0533 DEX        verringern, um Kopf nach außen zu schieben
0534 CLC        Carry-Flag setzen
0535 BCC $053B  unbedingter Sprung

0537 LDA $1C00  Control-Register laden
053A INX        erhöhen für Kopfbewegung nach außen
053B TXA        Wert in Akku schieben
053C AND #$03   ersten zwei Bits isolieren
053E STA $4B    und zwischenspeichern
0540 LDA $1C00  Control-Register erneut laden
0543 AND #$FC   Bits 0 und 1 löschen
0545 ORA $4B    und mit den errechneten Bits verknüpfen
0547 STA $1C00  und speichern
054A LDX #$09   $09 ist Zähler für
054C STX $4B    die Warteschleife,
054E DEC $4B    um dem Tonkopf
0550 BNE $054E  genügend Zeit
0552 DEX        zu geben,
0553 BNE $054E  positioniert zu werden
0555 RTS        Rücksprung
```

Sie werden sich sicher fragen, wie es möglich ist, daß sich der Tonkopf mittels dieser Routine bewegt. Für diese Bewegung sind die Bits 0 und 1 des Control-Registers zuständig. Der Kopf bewegt sich durch systematische Veränderung dieser Bits, wenn zusätzlich der Laufwerksmotor (Bit 3) eingeschaltet ist. Durch eine Veränderung der beiden Bits in der Folge 00/01/10/11/00 bewegt sich der Kopf nach innen, durch die Folge 00/11/10/01/00 bewegt er sich nach außen.

Hier ein Beispiel zum besseren Verständnis. Sind die Bits 0 und 1 wie in unserem Beispiel gelöscht und wird dann das erste Bit gesetzt, was der Bitfolge 01 entspricht, so bewegt sich der Kopf, wenn man ihm genug Zeit läßt, um eine Halbspur nach innen. Durch das Setzen beider Bits bewegt er sich nach außen. Wie Sie

sicher festgestellt haben, wird dieses Programm nicht über die Jobcodes, sondern direkt gestartet und läuft folglich auch nicht als Interrupt-Programm.

Zum Schluß noch ein Beispiel für das Starten eines Programms mittels des Jobcode $E0.

Das folgende Programm fährt den Tonkopf auf den eingestellten Track und liest dort den ersten Blockheader, den es findet. Die gelesenen Daten werden aus dem GCR-Format in die normalen Bitwerte gewandelt und in den Adressen $16 bis $1A gespeichert. Die Speicherung geschieht in der Reihenfolge:

```
$16 - Erste ID auf Diskette
$17 - Zweite ID auf Diskette
$18 - gelesener Track
$19 - gelesener Sektor
$1A - Prüfsumme über den Blockheader
```

Die GCR-Daten des Headers stehen ab $24

```
0500 JMP $0503  Sprung für den ersten Durchlauf unbedeutend
0503 LDA #$19   LOW-Byte der Einsprungadresse
0505 STA $0501  in den JMP-Befehl schreiben (JMP $0519)
0508 LDA #$01   Track, auf dem ausgeführt werden soll
050A STA $0A    in Speicher für Puffer zwei speichern
050C LDA #$00   Sektornummer (unerheblich)
050E STA $0B    speichern
0510 LDA #$E0   Jobcode $E0 (Programm im Puffer ausführen)
0512 STA $02    in Jobspeicher für Puffer zwei schreiben
0514 LDA $02    Jobspeicher lesen
0516 BMI $0514  verzweige wenn nicht beendet
0518 RTS        Rücksprung
```

Das auszuführende Interruptprogramm

```
0519 LDA #$03   Einspung wieder
051B STA $0501  normalisieren (JMP $0503)
051E LDX #$5A   90 Leseveruche
0520 STX $4B    im Zähler speichern
0522 LDX #$00   Zeiger auf 0 setzen
0524 LDA #$52   GCR-Codierung $08 (Headerkennzeichen)
0526 STA $24    in Arbeitsspeicher speichern
0528 JSR $F556  auf SYNC warten
052B BVC $052B  auf BYTE-READY beim Lesen warten
052D CLV        BYTE-READY wieder löschen
052E LDA $1C01  gelesenes Byte vom Port holen
0531 CMP $24    mit gespeichertem Header vergleichen
0533 BNE $0548  verzweige, wenn kein Blockheader gefunden
0535 BVC $0535  sonst auf BYTE-READY warten
0537 CLV        Leitung rücksetzen
0538 LDA $1C01  gelesenes Byte holen
053B STA $25,x  und in Arbeitsspeicher schieben
053D INX        Zeiger erhöhen
053E CPX #$07   schon ganzen HEADER gelesen?
0540 BNE $0535  verzweige, wenn noch nicht alle Zeichen
0542 JSR $F497  GCR-BYTEs in Bitform wandeln
0545 JMP $FD9E  Rücksprung aus dem Interrupt (ok)

0548 DEC $4B    Zähler für Fehlversuche verringern
054A BNE $0522  verzweige wenn weitere Versuche
054C LDA #$02   Fehlermeldung ($02=Blockheader nicht
054E JMP $F969  gefunden) ausgeben und Programm beenden
```

Das Programm wird ab $0500 gestartet und stellt den JMP $0503 auf JMP $0519 um. Daraufhin wird die Tracknummer übergeben und der Jobcode $E0 in den Jobspeicher $02 (Pufffer 2 - $0500) übergeben. Dieser Code wird nun von der interruptgesteuerten Jobschleife erkannt, die mit der Bearbeitung des Codes anfängt. Als erstes wird der Tonkopf auf den entsprechenden Track positioniert und das auszuführende Programm ab $0500 (für Puffer 2) gestartet. Da der erste Befehl jedoch JMP $0519 ist, wird der Programmteil, der vom Benutzer gestartet wurde,

übersprungen und der JMP Befehl wieder auf JMP $0503 zurückgesetzt. Das jetzt arbeitende Programm läuft interruptgesteuert. Das Hauptprogramm wartet bei $0514 bis $0516 darauf, daß eine Rückmeldung vom Interruptprogramm ankommt. Das Hauptprogramm muß immer mit einem RTS abgeschlossen werden, da sonst keine Meldung an den Computer geht und die Floppy sich aufhängt.

Das Interruptprogramm darf nicht mit einem RTS abgeschlossen werden. Am Ende dieses Programms muß wieder in die Jobschleife zurückgekehrt werden, welche die im Akku enthaltene Rückmeldung an den Jobspeicher übergibt. Nachdem das Hauptprogramm die Rückmeldung erhält, wird es mit dem RTS beendet. Mit $0545 JMP $FD9E wird eine $01 in den Akku geladen und danach ebenfalls mit JMP $F969 in die Jobschleife gesprungen.

Einer kleinen Erklärung bedarf es wohl noch bei

```
$0524 LDA #$52
```

Die $52 stellt die ersten 8 Bits einer im GCR-Format geschriebenen $08 dar, die bekanntlich für das Kennzeichen des Anfangs eines Blockheaders steht. Normalerweise kann nach dem Auffinden einer SYNC-Markierung nur eine $08 (Blockheader) oder $07 (Datenheader) gefunden werden. Da die GCR-Daten der beiden Zahlen sich jedoch schon nach den ersten 8 Bits unterscheiden, reicht es aus, nur diese zu vergleichen.

HEX	BIN	GCR
7	00000111	01010101 11 = $55+%11
8	00001000	01010010 01 = $52+%01

Wenn wir schon von GCR-Codierung sprechen: Wie kann man Daten, die man auf Disk schreiben will, in das GCR-Format und umgekehrt wieder ins Binärsystem wandeln? Für diese Fälle gibt es einige Routinen im DOS der Floppy.

$F78F Diese Routine wandelt den gesamten aktiven Puffer ins GCR-Format. Da bei der Codierung von 4 Bits zu 5 Bits selbstverständlich mehr Platz benötigt wird, werden die ersten GCS-Daten in dem Ausweichpuffer vom $01BB bis $01FF geschrieben. Die restlichen Daten stehen im aktiven Puffer.

$F5F2 Die Routine wandelt die GCR-Daten, die im Ausweichpuffer und im aktuellen Puffer stehen, wieder in den Binärcode zurück und speichert die Werte im aktuellen Puffer.

$F934 Wandelt den Blockheader, dessen Werte in den Speicherstellen $16 bis $1A stehen, in GCR-Daten um und speichert sie ab $24.

$F497 Wandelt den gelesenen Blockheader in Binärcode um. Die GCR-Daten stehen hierbei ab $24 und die Binärwerte ab $16 (siehe Beispielprogramm).

Das sollte als Einführung in die Programmierung der Floppy reichen. Zum Abschluß noch eine Beschreibung der Register der VIA 2, die von großem Interesse für die Programmierung der Floppy ist.

Wie funktioniert die Formatroutine?

Jeder von Ihnen wird schon Erfahrung mit der Formatroutine der Floppy gemacht haben, doch hier wollen wir sie einmal nicht aus der Sicht des Anwenders, sondern aus der Sicht des Programmierers betrachten.

Die Formatroutine, die bei $FAC7 beginnt, wird von der Routine ab $C8C6 mit dem Jobcode $E0 aufgerufen. Dazu wird von dieser Routine in die Speicherstellen $0600 bis $0602 ein 'JMP $FAC7' geschrieben und dieser mit dem $E0-Befehl gestartet.

Ab jetzt beginnt die eigentliche Formatroutine. Sie prüft die Speicherstelle $51, die den Track angibt, der gerade formatiert

wird. Sollte der Wert dieser Speicherstelle $FF sein, so erkennt das Programm, daß die Formatierung gerade begonnen wurde, und führt einen BUMP (Anschlag des Kopfes) aus. Im folgenden wird der Track genau vermessen, und aufgrund dieser Messung werden die Abstände zwischen den Sektoren festgelegt. Diese Messung ist auch der Grund dafür, daß das Formatieren einer Diskette so lange dauert. Die Sektoren werden nun mitsamt ihren Headern aufgetragen, womit auch schon die Formatierung eines Tracks beendet ist. Nun wird die aktuelle Tracknummer mit 36 verglichen. Sollte sie kleiner sein, verzweigt das Programm zur Trackpositionierung, von wo aus wieder in die Jobschleife gesprungen wird. Die Jobschleife prüft erneut, ob ein Jobbefehl ausgeführt werden soll, und erkennt den $E0-Job, der sich immer noch in $03 (Job-Speicher für Puffer 3) befindet. Das führt dazu, daß das Formatierungsprogramm ab $FAC7 durch den 'JMP'-Befehl ab $0600 erneut aufgerufen wird. Wenn man in die Formatroutine eingreifen will, reicht es aus, ab $0600 eine eigene Routine einzusetzen.

Das sollte als Einführung in die Programmierung der Floppy reichen. Viele weitere Erfahrungen und Anregungen können Sie durch häufiges und intensives Studium der DOS-Listings erhalten.

Zum Abschluß noch die Registerbeschreibung der Drive Control-Register.

6.1.4 Registerbeschreibung der VIA 2

```
$1C00 Drive-Control-Bus
      Bit 0 Steppermotorsteuerung
      Bit 1 Steppermotorsteuerung
      Bit 2 Laufwerksmotor;     Bit=0: Motor aus
      Bit 3 Laufwerk-LED;       Bit=0: LED aus
      Bit 4 Schreibschutz;      Bit=0: Lichtschranke unterbrochen
      Bit 5 Zur Einstellung der Bitrate zum Tonkopf
            (Speedflag)
      Bit 6 Bedeutung wie Bit 5
      Bit 7 SYNC-Signal beim Lesen gefunden; Bit=0: SYNC gefunden
```

$1C01 Port A Ein-/Ausgaberegister - Datenbus zum R/W-Kopf

$1C02 Datenrichtungsregister Port B

$1C03 Datenrichtungsregister Port A

$1C04 Timer 1 (LOW)

$1C05 Timer 1 (HIGH)

$1C06 Timer 1 (LOW Latch)
Beim Schreib- oder Lesezugriff bleibt der Wert des Timers unverändert. Der Wert wird in diesem Zwischenspeicher gespeichert und im FREE-RUNNING-MODE ($1C11 Bit 6=1) bei jedem Null-Durchgang in den Zähler automatisch übernommen.

$1C07 Timer 1 (HIGH Latch)
Siehe $1C06

$1C08 Timer 2 (LOW)

$1C09 Timer 2 (HIGH)

$1C0A Schieberegister

$1C0B Hilfssteuerregister
Bit 0 PA (Latch-Enable-Disable)
Bit 1 PB (0=Disable 1=Enable Latching)
Bit 2 Schiebesteuer-Bit
Bit 3 Schiebesteuer-Bit
Bit 4 Schiebesteuer-Bit
Bit 5 T2 Timercontrol (0=Zeitlicher Interrupt
1=Abwärtszählen mit Signal am Anschuß PB6)
Bit 6 T1 Timercontrol
Bit 7 T1 Timercontrol

```
$1C0C Peripheristeuerregister (PCR)
      Bit 0 CA1 Interruptsteuerung (0=Negative Flanke des
            Signals/1=Positive Flanke)
      Bit 1 CA2 Control
      Bit 2 CA2 Control
      Bit 3 CA2 Control
      Bit 4 CB1 Interruptsteuerung siehe Bit 0
      Bit 5 CB2 Control
      Bit 6 CB2 Control
      Bit 7 CB2 Control

$1C0D Interrupt-Flag-Register
      Dieses Register signalisiert das Eintreffen von
      Ereignissen durch setzen des entsprechenden Bits
      dieses Registers.
      Bit 0 Aktive Flanke an CA2
      Bit 1 Aktive Flanke an CA1
      Bit 2 acht Schiebeimpulse von SR ($1C0A)
      Bit 3 Aktive Flanke an CB2
      Bit 4 Aktive Flanke an CB1
      Bit 5 Timer-Unterlauf von Timer 2
      Bit 6 Timer-Unterlauf von Timer 1
      Bit 7 Wird gesetzt wenn eines der Bits sowohl in
            diesem Register, als auch in $1C0E gesetzt ist.

$1C0E Interrupt Enable Register
      Die Bits dieses Registers korrespondieren mit den
      Bits aus $1C0D. Ist in diesem Register ein Bit
      gesetzt und der entsprechende Zustand, der in $1C0D
      angezeigt wird, erfüllt, wird ein IRQ ausgeführt.

$1C0F Datenregister A
      Dieses Register spiegelt den Inhalt aus $1C00, jedoch
      ohne Handshake-Betrieb.
```

6.2 Die Formatroutine

Durch Änderung der Formatroutine lassen sich einige gut zu gebrauchende Effekte erzielen. Es ist möglich, einen einzelnen

Track oder die Spuren 36 bis 41 zu formatieren sowie sämtliche Blockheader-Parameter zu verändern.

6.2.1 Formatieren eines einzelnen Tracks

Im Kapitel 6.1.3 wurde schon allgemein auf die Arbeitsweise der Formatroutine eingegangen. Dort haben Sie erfahren, daß die Formatroutine vor der Formatierung eines Tracks jedesmal neu über den 'JMP $FAC7' bei $0600 aufgerufen wird. An dieser Stelle greifen wir nun ein und zwingen die Routine, nur einen Track zu formatieren. Für dieses Vorhaben reicht das folgende kleine Programm, das mit dem Job-Code 'E0' im Job-Speicher $03 aufgerufen werden muß.

```
0600 JMP $0603   Der JMP wird vom Programm noch verändert
0603 LDA $0C     Nummer des zu formatierenden Tracks holen
0605 STA $51     und übergeben
0607 LDA #$0F    JMP-Befehl auf $060F zeigen lassen
0609 STA $0601   um beim erneuten Aufruf die Formatierung
                 zu beenden
060C JMP $FAC7   zur Formatroutine springen
060F JMP $FD9E   beim zweiten Aufruf Programm beenden
```

Hier wird das Programm mit einem 'B-E'-Befehl gestartet

```
0612 LDA #$01    Nummer des zu formatierenden Tracks laden
0614 STA $0C     und an den Job übergeben
0616 LDA #$E0    Job-Befehl E0 (Programm aufführen) laden
0618 STA $03     und in den Job-Speicher schreiben
061A RTS         Rücksprung
```

Als nächstes folgt ein BASIC-Programm, das es erlaubt, die entsprechende Nummer des zu formatierenden Tracks einzugeben. Zu Beginn des Programms wird initialisiert, damit die Formatroutine den Track auch mit der richtigen ID formatiert. Sollte dies nicht der Fall sein, läßt sich ein Block auf diesem

Track nicht mehr ohne eigene Software lesen. Ein so "zerstörter" Track ist jedoch für Kopierprogramme des heutigen Standards kein Problem. Er kann lediglich genutzt werden, um Knackern den Zugriff auf den Block, auf dem die Kopierschutzabfrage steht, zu erschweren. Jetzt aber das angekündigte Programm.

```
OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 INPUT "WELCHER TRACK";T
110 IF T >35 OR T<1 THEN100
120 PRINT#1,"M-W"CHR$(19)CHR$(6)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(18)CHR$(6)
140 PRINT#1,"M-R"CHR$(3)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A>127 THEN130
170 IFA =1 THENPRINT"FORMATING OK":END
180 PRINT"FORMAT ERROR":END
302 DATA 76,  3,  6,165, 12,133, 81,169, 15,141,  1,  6, 76,199,250, 76,
158,253
304 DATA169,  1,133, 12,169,224,133,  3, 96,  -1
```

Wie schon gesagt, wird als erstes initialisiert. Daraufhin werden die Daten in die Floppy ab $0600 geschickt. Nachdem die Track-Nummer eingegeben wurde, wird sie ebenfalls zur Floppy geschickt und das Programm ab $0612 mit dem 'M-E'-Befehl gestartet. Im Anschluß daran wird auf die Rückmeldung im Job-Speicher $03 gewartet und nach ihrem Erhalt die entsprechende Meldung ausgegeben.

Es ist nicht möglich, mit diesem Programm die Spuren oberhalb von 35 zu formatieren, was nicht von der Abfrage in Zeile 110, sondern von dem DOS der Floppy abhängt.

Will man Track 18 formatieren, so stößt man auf etwas größere Probleme. War dieser vor der Formatierung zerstört, so daß es nicht möglich ist zu initialisieren, muß vor der Formatierung die ID 'per Hand' angegeben werden. Für diesen Zweck müssen Sie

mit dem 'M-W'-Befehl die ID in die Floppy-Speicherstellen $12 (18) und $13 (19) schreiben. Sie werden sich vielleicht fragen, wie Sie die ID Ihrer Diskette erfahren sollen, falls Sie das Directory nicht mehr laden können. Für diesen Fall können Sie das im vorherigen Kapitel beschriebene Programm verwenden, das den nächsten Blockheader, den es findet, liest und decodiert. Die ID des Blockheaders steht dann in Speicherstelle $16 (22) und $17 (23), von wo Sie sie ohne weiteres auslesen können.

Nach dem Formatieren des Tracks 18 kann man jedoch nicht gleich Programme auf der Diskette speichern. Zuvor muß noch die BAM und das Directory erstellt und auf Diskette gespeichert werden. Dies erledigt für uns zum größten Teil eine Routine im DOS der Floppy. Wir müssen ihr lediglich den Namen unserer Diskette übergeben. Es ist nicht möglich, die Diskette 'soft' zu formatieren (Formatierungsbefehl ohne ID-Angabe), da dieser Befehl nur angewendet werden kann, wenn die BAM schon vorher vorhanden war. Das folgende Programm wird in der Floppy ab $0400 mit einem 'M-E'-Befehl gestartet. Die Daten für den Diskettennamen müssen in diesem Fall ab $0420 im Speicher stehen und mit einem Komma abgeschlossen werden.

```
0400 LDX #$10     16 Buchstaben des Namens
0402 LDA $0420,x  vom $0420 in den
0405 STA $0200,x  Befehlspuffer schieben
0408 DEX          Zähler verringern
0409 BPL $0402    verzweige, wenn noch nicht fertig
040B LDA #$10     16 Buchstaben für den Namen
040D STA $0274    in Puffer zulassen
0410 JMP $EE40    Befehl ausführen, Rücksprung
```

Das folgende BASIC-Programm erledigt alle Aufgaben für Sie. Der Name wird der Floppy übergeben, und daraufhin wird das Programm ausgeführt.

```
0 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
30 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 INPUT "DISKNAME";N$:N$=N$+","
110 FORN=0TO LEN(N$)-1
120 PRINT#1,"M-W"CHR$(32+N)CHR$(4)CHR$(1)MID$(N$,N+1,1): NEXT
130 PRINT#1,"M-E"CHR$(0)CHR$(4)
302 DATA162, 16,189, 32,  4,157,  0,  2,202, 16,247,169, 16,141,116,  2,
 76, 64
304 DATA238,-1
```

Das Programm fügt selbstständig das Komma am Ende des Namens an.

Mit diesen Programmen ist es Ihnen also möglich, einen einzelnen Track zu formatieren, solange er nicht oberhalb von Spur 35 liegt.

6.2.2 Formatieren der Spuren 36 bis 41

Es ist der Floppy ohne weiteres möglich, ihren R/W-Kopf auf die Spuren oberhalb der Spur 35 zu positionieren. Wollen Sie diese Spuren für die Ablage von Daten verwenden, so stoßen Sie noch auf ein Problem. Sie können den R/W-Kopf zwar ohne Probleme auf die Spuren oberhalb von 35 fahren, dort jedoch nicht ohne weiteres lesen, denn die Tabelle, aus der sich das DOS die Anzahl der Sektoren pro Track und den Speed der entsprechenden Spur holt, gilt nur für die Spuren unterhalb von 36. Wir müssen deswegen sowohl beim Lesen als auch beim Schreiben den Speed und die Anzahl der Sektoren selber übergeben. Damit steht dem Formatieren der Spuren 36 bis 41 nichts mehr im Wege. Zu beachten ist lediglich, daß die Formatroutine nach der Formatierung eines Tracks erkennt, daß sie sich oberhalb von Spur 35 befindet und den Vorgang daraufhin abbricht. Wie müssen somit diese Routine vor der Formatierung eines weiteren Tracks erneut starten, was bei der Formatierung unter-

halb von 36 nicht gemacht werden mußte. Im folgenden zeigen wir das Programm, das diese Spuren formatiert. Es wird in der Floppy mit dem 'M-E'-Befehl ab $0415 gestartet.

```
0400 LDA #$11    Anzahl der Sektoren für die
0402 STA $43     Spuren über 35 übergeben
0404 LDA $1C00   Control-Port laden
0407 AND #$9F    und Speed für diese Tracks
0409 ORA #$00    einstellen (kann hier geändert werden)
040B STA $1C00   Wert speichern
040E LDA $08     Track-Nummer laden
0410 STA $51     und die Nummer übergeben
0412 JMP $FAC7   Formatroutine anspringen
```

Das Programm wird hier gestartet

```
0415 JSR $D042   initialisieren
0418 LDA #$24    Track, bei dem die Formatierung beginnt
041A STA $37     laden $24 (36) und zwischenspeichern
041C STA $08     Track-Nummer an Job übergeben
041E LDA #$E0    Job-Code E0 für Programm ausführen
0420 STA $01     in Job-Speicher schreiben
0422 LDA $01     Rückmeldung erwarten
0424 BMI $0422   verzweige, wenn Job nicht abgearbeitet
0426 CMP #$02    Rückmeldung auf Fehler prüfen
0428 BCS $0432   verzweige, wenn Fehler aufgetreten
042A INC $37     Track-Nummer erhöhen
042C LDA $37     und vergleichen,
042E CMP #$2A    ob Spur 41 schon überschritten
0430 BNE $041A   wenn nicht, nächsten Track formatieren
0432 RTS         Rücksprung
```

Im folgenden ein BASIC-Programm, das Ihnen die Eingabe des Anfangs- und des End-Tracks erlaubt. Der End-Track sollte nicht höher als Spur 42 liegen, da der R/W-Kopf sonst anschlägt und sich dadurch das Laufwerk dejustieren kann. Track 42 schaffen die meisten Floppys noch gerade zu formatieren, sofern

eine qualitativ hochwertige Diskette verwendet wird. Diesen Track erreicht bis heute kein von uns bekanntes Kopierprogramm.

```
10 OPEN 1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IF SU <> 5381 THENPRINT"FEHLER IN DATAS":STOP
102 INPUT "STATRTRACK >35 ";S
105 INPUT "ENDTRACK <42 ";E
120 PRINT#1,"M-W"CHR$(25)CHR$(4)CHR$(1)CHR$(S)
125 PRINT#1,"M-W"CHR$(47)CHR$(4)CHR$(1)CHR$(E+1)
130 PRINT#1,"M-E"CHR$(24)CHR$(4)
140 PRINT#1,"M-R"CHR$(1)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A>127 THEN140
170 IFA =1 THENPRINT"FORMATING OK":END
180 PRINT"FORMAT ERROR":END
302 DATA169, 17,133, 67,173,  0, 28, 41,159,  9,  0,141,  0, 28,165,  8,
133, 81
304 DATA 76,199,250, 32, 66,208,169, 36,133, 55,133,  8,169,224,133,  1,
165,  1
306 DATA 48,252,201,  2,176,  8,230, 55,165, 55,201, 42,208,232, 96, -1
```

Wir können uns jetzt mit Hilfe dieses Programms die Spuren 36 bis 41 formatieren und sie somit für die Speicherung von Daten vorbereiten. Doch die Speicherung und das Laden von Daten auf diesen Spuren ist nicht so leicht wie auf den normal verwendeten Tracks. Wie schon gesagt, kann das DOS für diese 'illegalen' Spuren den Speed und die Sektorenanzahl nicht richtig bestimmen. Aus diesem Grund kann man weder mit den USER-Befehlen 'U1' (Block laden) und 'U2' (Block speichern) noch mit den Job-Codes '80' (Block laden) sowie '90' (Block speichern) arbeiten. Die einzige Möglichkeit, die uns bleibt, ist, über den Job-Code 'E0' ein Programm aufzurufen, das den Speed und die

Anzahl der Sektoren setzt und danach die Block-Lese- oder Schreibroutine anspringt. Das Programm, das einen Block liest, sieht dann folgendermaßen aus:

```
0600 LDA $1C00    Control-Port laden
0603 AND #$9F     Bits für Speed löschen
0605 ORA #$00     und mit neuem Speed verknüpfen
0607 STA $1C00    Wert wieder speichern
060A LDA #$11     Anzahl der Sektoren laden
060C STA $43      und übergeben
060E LDA #$05     HIGH-Byte der Pufferadresse, in den
0610 STA $31      geladen werden soll, angeben.
0612 JMP $F4D1    zur Laderoutine springen
```

Das Programm wird hier gestartet

```
0615 LDA #$24     zu ladenden Track für
0617 STA $0C      den Job übergeben
0619 LDA #$00     Sektor-Nummer an
061B STA $0D      den Job übergeben
061D LDA #$E0     Job-Code für Programm ausführen in den
061F STA $03      Job-Speicher schreiben
0621 LDA $03      Rückmeldung abwarten
0623 BMI $0621    verzweige, wenn noch nicht fertig
0625 RTS          Rücksprung
```

Das Programm wird ab $0615 mit dem 'M-E'-Befehl gestartet und lädt den angegebenen Sektor in Puffer 2 ($0500). Auch hier muß vor dem Laden des Blocks der Speed und die Anzahl der Sektoren übergeben werden, worauf die Routine zum Lesen eines Blocks angesprungen wird. Wenn Sie einen Block auf die oberen Spuren sichern wollen, so reicht es aus, den 'JMP $F4D1' bei Adresse $0612 in ein 'JMP $F575' zu ändern. Das nachfolgende BASIC Programm erlaubt Ihnen die Eingabe des Tracks und des Sektors, von dem Sie laden wollen.

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 3608 THEN PRINT"ERROR IN DATAS":STOP
105 INPUT "WELCHER TRACK";T
110 INPUT "WELCHER SEKTOR";S
120 PRINT#1,"M-W"CHR$(22)CHR$(6)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(26)CHR$(6)CHR$(1)CHR$(S)
130 PRINT#1,"M-E"CHR$(21)CHR$(6)
135 FORN=1TO 500:NEXT
140 PRINT#1,"M-R"CHR$(3)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A>127 THEN130
170 IFA =1 THENPRINT"OK":END
180 PRINT"ERROR":END
302 DATA173,  0, 28, 41,159,  9,  0,141,  0, 28,169, 17,133, 67,169,  5,
133, 49
304 DATA 76,209,244,169, 36,133, 12,169,  0,133, 13,169,224,133,  3,165,
  3, 48
306 DATA252, 96, -1
```

Der gelesene Block wird, wie schon gesagt, in Puffer 2 ($0500) geschrieben, von wo aus er in den Computer geholt werden kann. Um mit diesem Programm einen Block zu schreiben, ist es notwendig, die Zeile 304 und die Prüfsumme über die Daten zu ändern. Anstatt 304 DATA 76,209,244 ... muß hier 304 DATA 76,117,245 ... stehen. Die Prüfsumme in Zeile 100 muß dann '3517' anstelle von '3608' lauten. Die Daten, die auf den eingegebenen Track und Sektor geschrieben werden, werden natürlich auch wieder aus Puffer 2 ($0500) geholt. Es ist nicht möglich, mit diesem Programm Blöcke auf den Tracks unterhalb von Track 31 zu schreiben, da diese Tracks mit einem anderen Speed beschrieben werden.

6.2.3 Doppelte Spuren

Ein Kopierschutzsystem, daß sich weniger durch seine Kompliziertheit als durch seine Raffiniertheit auszeichnet, soll nun beschrieben werden. Es ist schwer auszumachen, weil keine Fehler auf der Diskette zu erkennen sind. Bei diesem Kopierschutz handelt es sich auch, wie Sie sich sicher denken können, um eine Änderung der Formatroutine. Der Gedanke, der hinter diesem Kopierschutz steckt, ist recht einfach. Die unteren Spuren werden doppelt angelegt. Das heißt, daß die Tracks auf Diskette anstelle der gewohnten Reihenfolge Track 1, Track 2, Track 3 usw. die Folge Track 1, Track 2, Track 1, Track 2, Track 3, Track 4, Track 5 usw. aufweisen. Sie erkennen, daß der Track 1 und Track 2 doppelt auf Diskette vorhanden sind.

Sie werden sich vielleicht fragen, was das in Hinblick auf Kopierschutz bedeutet. Die Antwort ist einfach, wenn man sich daran erinnert, wie das DOS auf die einzelnen Tracks zugreift. Als erstes wird "nachgesehen", auf welchem Track sich der R/W-Kopf gerade befindet. Danach wird die Differenz zwischen der jetzigen und der zu ereichenden Spur errechnet und der R/W-Kopf um die entsprechende Anzahl von Spuren verschoben. Der Kopf erreicht nach diesem Prinzip somit nie auf normalem Wege die gedoppelten, unterhalb der echten liegenden Tracks 1 und 2. Nach genau diesem Prinzip arbeitet auch die Positionierung des R/W-Kopfes bei den Kopierprogrammen. Für diese ist es nicht erkennbar, daß unterhalb des gefundenen Track 1 sich noch 1, 2 oder mehr Tracks befinden. Es gibt keine Probleme beim Arbeiten mit der Diskette, auch wenn der Kopf einmal einen 'Bump' (Anschlag des Kopfes) machen sollte, denn er fährt zu der vor dem Anschlag errechneten Spur, erkennt, daß er sich noch um einige Spuren zu tief befindet, und fährt den Kopf an die richtige Position.

Wie schon anfangs gesagt, ist es sehr einfach, eine Diskette derart zu formatieren. Man muß nur vorher den Kopf um die eingestellte Anzahl von Spuren nach innen verschieben. Von dieser Verschiebung merkt das DOS nichts und "denkt", es würde den Kopf auf Spur 1 fahren. Stattdessen beginnt es jedoch die Formatierung einige Spuren höher.

Nach der ausführlichen Erklärung des Prinzips wollen wir zur Praxis übergehen und ein solches Programm ausprobieren. Das folgende Maschinenspracheprogramm, das wie immer in der Floppy gestartet wird, erfüllt diesen Zweck. Es wird ab $0450 mit einen 'M-E'-Befehl gestartet

```
0400 JMP $0403   für ersten Aufruf belanglos
0403 LDA #$17    LOW-Byte für die Änderung des 'JMP'
0405 STA $0401   LOW-Byte des 'JMP' auf $0417 ändern
0408 LDA #$04    Anzahl der Halbspuren, die verschoben
040A STA $37     werden, zwischenspeichern
040C JSR $041A   Kopf um eine Halbspur verschieben
040F DEC $37     Zähler verringern
0411 BNE $040C   verzweige, wenn nicht genug verschoben
0413 LDA #$01    Nummer des Tracks, ab dem formatiert
0415 STA $51     werden soll, übergeben
0417 JMP $FAC7   zur Formatroutine springen
```

Kopf-Verschieberoutine

```
041A LDX $1C00   Control-Port laden
041D INX         erhöhen für Kopfbewegung nach innen
041E TXA         Wert in Akku schieben
041F AND #$03    erste zwei Bits isoliern
0421 STA $4B     und zwischenspeichern
0423 LDA $1C00   Control-Port laden
0426 AND #$FC    Bit 0 und 1 löschen
0428 ORA $4B     mit den errechneten Bits verknüpfen
042A STA $1C00   und Speichern
042D LDX #$00    Zähler für Warteschleife setzen, um
042F LDY #$05    dem Kopf genügend Zeit zu lassen
0431 DEX         positioniert zu werden
0432 BNE $0431   verzweige, wenn LOW-Byte nicht abgelaufen
0434 DEY         HIGH-Byte verringern
0435 BNE $0431   springe, wenn HIGH-Byte nicht abgelaufen
0437 RTS         Rücksprung
```

Das Programm wird hier gestartet

```
0438 LDA #$12     Track 18 für Job
043A STA $08      übergeben
043C LDA #$E0     Job-Code 'E0' laden
043E STA $01      und in Job-Puffer schreiben
0440 LDA $01      Rückmeldung holen
0442 BMI $0440    verzweige, wenn noch nicht fertig
0444 CMP #$02     Rückmeldung auf Fehler prüfen
0446 BCS $0459    verzweige, wenn Fehler erkannt
0448 LDX #$10     Disknamen für BAM 16 Buchstaben erlauben
044A STX $0274    Anzahl übergeben
044D LDA $0480,X  Namen von $0480 in
0450 STA $0200,X  den Befehls-Puffer schreiben
0453 DEX          Zähler verringern
0454 BPL $044D    verzweige, wenn nicht alle Buchstaben
0456 JMP $EE40    BAM schreiben, Rücksprung
0459 RTS          Rücksprung bei Fehler
```

Im Anschluß an dieses Programm folgt ein BASIC-Programm, das uns die Eingabe der Anzahl der Tracks ermöglicht, die bei der Formatierung doppelt anglegt werden. Um diese Spezialformatierung nicht zu auffällig zu machen, sollte man nicht mehr als zwei Tracks höher mit der Formatierung beginnen. Damit dieses Programm richtig arbeitet, müssen die zu behandelnden Disketten schon fehlerfrei formatiert gewesen sein. Die neue Diskette wird mit der gleichen ID formatiert, mit der sie zuvor formatiert wurde. Doch nun zu dem angekündigten BASIC-Programm, das das Maschinensprachprogramm als erstes zur Floppy schickt, um es nach der Eingabe der Parameter dort zu starten.

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IF SU <> 9284 THENPRINT"FEHLER IN DATAS":STOP
```

```
102 INPUT "WIEVIELE TRACKS VERSCHIEBEN <8 : ";AN
104 INPUT "DISKNAME";N$:N$=N$+","
106 FORN=0TO LEN(N$)-1
108 PRINT#1,"M-W"CHR$(128+N)CHR$(4)CHR$(1)MID$(N$,N+1,1):   NEXT
120 PRINT#1,"M-W"CHR$(9)CHR$(4)CHR$(1)CHR$(AN*2)
130 PRINT#1,"M-E"CHR$(56)CHR$(4)
140 PRINT#1,"M-R"CHR$(1)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A>127 THEN140
170 IFA =1 THENPRINT"FORMATING OK":END
180 PRINT"FORMAT ERROR":END
402 DATA 76,  3,  4,169, 23,141,  1,  4,169,  4,133, 55, 32, 26,  4,198,
 55,208
404 DATA249,169,  1,133, 81, 76,199,250,174,  0, 28,232,138, 41,  3,133,
 75,173
406 DATA  0, 28, 41,252,  5, 75,141,  0, 28,162,  0,160,  5,202,208,253,
136,208
408 DATA250, 96,169, 18,133,  8,169,224,133,  1,165,  1, 48,252,201,  2,
176, 17
410 DATA162, 16,142,116,  2,189,128,  4,157,  0,  2,202, 16,247, 76, 64,
238, 96
412 DATA -1
```

Nachdem wir uns mit dem Auftragen des Kopierschutzes beschäftigt haben, wollen wir uns um die Abfrage desselben kümmern. Natürlich kann man nicht einfach mittels eines 'U1'-Befehls der Floppy mitteilen, daß man gedenkt, einen Block auf Track 1 zu lesen, der unterhalb des vermuteten Track 1 liegt. Um dieses zu können, ist es wieder notwendig, mit einem Maschinensprachprogramm in der Floppy zu arbeiten. Dieses Programm fährt den R/W-Kopf auf den "normalen" Track 1, verschiebt mit Hilfe der schon verwendeten Routine den Kopf der Floppy um eine Spur nach außen und sucht erneut die Spur 1. Das erneute Suchen ist wichtig, da das DOS "denkt", es würde sich noch auf Track 1 befinden. Würde man einen anderen Track suchen lassen, würde der Kopf auf diese Spur fahren, die jedoch nicht unserer doppelten entspricht. Sucht man jedoch die Spur 1, auf der sich der Kopf vermeintlich befindet, so stellt das DOS fest, sofern mehr als eine Spur höher formatiert wurde,

daß der Kopf sich auf einer höheren Spur befindet, und fährt den Kopf noch weiter nach unten, bis er die doppelte Spur 1 findet und sie als die richtige erkennt.

Nach dieser Prozedur kann man ohne Probleme die Blöcke auf diesen doppelten Spuren mit dem 'U1'-Befehl lesen oder schreiben. Das folgende Maschinensprachprogramm leistet das gerade besprochene. Es wird ab $0627 in der Floppy gestartet.

```
0600 JSR $0609   R/W-Kopf um eine
0603 JSR $0609   Spur verschieben (2 mal eine Halbspur)
0606 JMP $FD9E   Rücksprung

0609 LDX $1C00   Control-Register laden
060C DEX         verringern, um Kopf nach außen zu fahren
060D TXA         Wert in Akku schieben
060E AND #$03    und die ersten zwei Bit isolieren
0610 STA $4B     Wert zwischenspeichern
0612 LDA $1C00   Control-Register laden
0615 AND #$FC    Bit 0 und 1 löschen
0617 ORA $4B     mit gespeichertem Wert verknüpfen
0619 STA $1C00   und Bewegung ausführen
061C LDX #$00    Werte für die
061E LDY #$05    Zeitschleife laden
0620 DEX         Zähler verringern
0621 BNE $0620   verzweige, wenn nicht abgelaufen
0623 DEY         HIGH-Byte des Zählers verringern
0624 BNE $0620   verzweige, wenn nicht abgelaufen
0626 RTS         Rücksprung
```

Das Programm wird hier gestartet

```
0627 JSR $D042   Disk initialisieren
062A LDA #$01    Track auf den der Kopf gefahren wird
062C STA $0C     übergeben
062E LDA #$E0    Job-Code 'E0' laden
0630 STA $03     und in Job-Speicher schreiben
0632 LDA $03     Rückmeldung erwarten
0634 BMI $0632   verzweige, wenn noch keine Rückmeldung
```

```
0636 LDA #$01     Nummer für doppelten Track 1 laden
0638 STA $0A      und übergeben
063A LDA #$00     Sektor 0 laden
063C STA $0B      und übergeben
063E LDA #$80     Job-Code '80' (Block lesen) laden
0640 STA $02      und übergeben
0642 LDA $02      auf Rückmeldung warten
0644 BMI $0642    verzweige, wenn keine Rückmeldung
0646 RTS          Rücksprung
```

Sie können anhand des Programms das Prinzip noch einmal nachvollziehen. Das Programm lädt Block 1 des doppelten (unteren) Tracks automatisch in Puffer 2 ($0500). Hier nun der dazugehörige BASIC-Loader.

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
30 SU=SU+X:PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IF SU<>7037THENPRINT"FEHLER IN DATAS":STOP
130 PRINT#1,"M-E"CHR$(39)CHR$(6)
302 DATA 32,  9,  6, 32,  9,  6, 76,158,253,174,  0, 28,202,138, 41,  3,
133, 75
304 DATA173,  0, 28, 41,252,  5, 75,141,  0, 28,162,  0,160,  5,202,208,
253,136
306 DATA208,250, 96, 32, 66,208,169,  1,133, 12,169,224,133,  3,165,  3,
 48,252
308 DATA169,  1,133, 10,169,  0,133, 11,169,128,133,  2,165,  2, 48,252,
 96, -1
```

Wie schon erwähnt, können Sie nach dem Aufruf des Programms mit den normalen Block-Lese- und -Schreibbefehlen auf die Blöcke der unteren Tracks zugreifen. Wollen Sie nach Beendigung der Kopierschutzabfrage wieder auf die "richtigen" Tracks zugreifen, reicht es aus, die Diskette zu initialisieren oder einen nicht doppelt vorhandenen Track zu lesen.

6.2.4 Änderung der Headerparameter: Read-Errors

Ein weiteres sehr sicheres Kopierschutzsystem läßt sich auch mit einer anderen Änderung der Formatroutine erreichen. In diesem Fall handelt es sich um die Änderung der Blockheader-Parameter. Wenn Sie schon etwas über die Änderung der Header-Parameter gelesen oder es schon selbst gemacht haben, werden Sie wahrscheinlich sehr skeptisch sein, ob man mit einem solchen Eingriff einen wirklich guten Kopierschutz erstellen kann. Aber lassen Sie sich überraschen.

Die einfache Blockheader-Manipulation dient nur dem Zweck, Daten für den Knacker schwerer zugänglich zu machen, was auch nicht zu verachten ist. Den Kopierprogrammen heutigen Standards ist es jedoch ein leichtes, diese Blöcke zu kopieren.

Bevor wir besprechen, wie man diese veränderten Parameter auf Diskette aufträgt, wollen wir Sie jedoch erst noch weiter in die Arbeitsweise der Formatroutine einführen um die nachfolgenden Programme auch verstehen zu können.

Wir hatten schon gesagt, daß die Formatroutiene bei $FAC7 im Speicher der Floppy beginnt und auch dort aufgerufen wird. Nachdem der R/W-Kopf auf dem angegebenen Track positioniert wurde, wird die Spur genau vermessen, um festzustellen, wie groß die Lücke zwischen den einzelnen Sektoren sein soll. Diese Zahl darf nicht kleiner als 4 sein. Nach der Errechnung wird die Länge der Lücke in $0626 zwischengespeichert. Nach dieser Speicherung beginnt der zweite Teil der Formatroutine ab $FC36. Hier werden jetzt die Blockheader für die zu formatierenden Sektoren vorbereitet. Zu diesem Zweck werden die Parameter aus der Zeropage der Floppy geholt. Der Blockheader-Code $08 steht in $39, die Track-Nummer wird aus der Adresse $51 geholt. Die ID steht in $12, $13, und die zwei Lückenwerte sind $0F. Sie werden direkt geladen und sind, wie auch die Track-Nummer und die Sektornummer, nicht veränderbar. Daraufhin wird die Prüfsumme für den Blockheader berechnet.

Diese 8 Daten werden ab $0300 hintereinander liegend für jeden Header abgelegt und gemeinsam in das GCR-Format umgerechnet. Die Anzahl der Sektoren wird beim Erstellen der Blockheader in $0628 gespeichert. Vor der Umrechnung steht im Y-Register die Anzahl der Bytes, die zu den Blockheadern gehören.

Nach dieser Arbeit beginnt erst die eigentliche Formatierung, was bedeutet, daß die Blöcke mit ihren Blockheadern und alle SYNC-Markierungen auf Diskette geschrieben werden. Nach der Formatierung des Tracks wird die sich in $51 befindende Track-Nummer überprüft. Sollte sie höher oder gleich $24 (56) sein, wird die Formatierung beendet. Soweit die nähere Erläuterung der Formatroutine, die jedoch zum Verständnis des folgenden Programms nötig ist.

Um die Header-Parameter zu ändern, wollen wir diese nicht von der Formatroutine erstellen lassen, sondern sie selber im Computer erstellen und dann an die richtige Stelle in der Floppy schreiben. Natürlich dürfen wir dann die Formatroutine nicht von Anfang an starten, sondern erst dort einspringen, wo die Routine die Header-Parameter in das GCR-Format wandelt. Um die Formatroutine dort starten zu können, müssen auch die richtigen Parameter an die Routine übergeben werden, die sonst bis zu diesem Punkt von der Formatroutine selbst errechnet worden wären, wie beispielsweise die Länge der Lücke zwischen den Sektoren.

Alle diese Aufgaben erledigt für Sie das folgende etwas längere Programm. Mit ihm ist es auf komfortable Weise möglich sämtliche Blockheader-Parameter zu ändern und somit einzelne Sektoren vor dem Zugriff anderer zu schützen oder einen guten Kopierschutz aufzutragen.

```
10 A=49152
20 READ X:IF X=-1 THEN40
30 POKEA,X:A=A+1:SU=SU+X:GOTO20
40 IF SU <> 17072 THENPRINT"FEHLER IN DATAS":STOP
50 PO=29:PRINTCHR$(147):PRINT:PRINT:PRINT
60 PRINT"    SPEZIALFORMATIERUNG EINES TRACKS"
```

```
70 PRINT:PRINT:INPUT "    GEBEN SIE DEN TRACK AN";T
80 SE=21:L=9:SP=3
90 IF T>17 THENSE=19:L=9:SP=2
100 IF T>24 THENSE=18:L=10:SP=1
110 IF T>30 THENSE=17:L=11:SP=0
120 IF T<1 OR T>41 THEN PRINTCHR$(145)CHR$(145)CHR$(145);: GOTO70
130 PRINT:PRINT"    BITTE LEGEN SIE DIE DISKETTE EIN"
140 GET A$:IF A$=""THEN140
150 OPEN1,8,15,"I"
160 PRINT#1,"M-R"CHR$(18)CHR$(0)CHR$(2)
170 GET#1,I1$,I2$
180 PRINT:PRINT"    ID 1 DER DISKETTE IST ";ASC(I1$+CHR$(0))
190 PRINT:PRINT"    ID 2 DER DISKETTE IST ";ASC(I2$+CHR$(0))
200 PRINT:PRINT"    NEUE ID 1 FUER DEN TRACK  ";ASC (I1$+CHR$(0));:GOSUB
840
210 POKE249,EI
220 PRINT:PRINT"    NEUE ID 2 FUER DEN TRACK  ";ASC(I2$+CHR$(0));:GOSUB8
40
230 POKE250,EI
240 POKE248,T:POKE 49285,T
250 POKE251,SE:POKE 49257,SE
260 POKE 49264,L
270 POKE49249,SE*8
280 SYS49216
290 PRINT:PRINT"    HEADER-MANIPULATION (J/N)?"
300 GET A$:IFA$=""THEN300
310 IF A$="N"THEN630
320 PRINTCHR$(147);
330 PRINTCHR$(19)CHR$(17)CHR$(17);"  WELCHER HEADER 0 -";SE-1;:INPUTH
340 IF H<0 OR H> SE-1 THEN 330
350 X=49664+H*8
360 PO=20:PRINT:PRINT"  HEADER-POSITION ";H
370 PRINT:PRINT:PRINT"  HEADER-KENNZEICHEN  ";PEEK(X);:GOSUB
840:POKEX,EI
380 PRINT"  ERSTE  ID          ";PEEK(X+5);:GOSUB 840:POKEX+5,EI
390 PRINT"  ZWEITE ID          ";PEEK(X+4);:GOSUB 840:POKEX+4,EI
400 PRINT"  TRACK-NUMMER        ";PEEK(X+3);:GOSUB840:POKEX +3,EI
410 PRINT"  SEKTOR-NUMMER       ";PEEK(X+2);:GOSUB840:POKEX +2,EI
420 PRINT"  LUECKE 1           ";PEEK(X+6);:GOSUB840:POKEX +6,EI
430 PRINT"  LUECKE 2           ";PEEK(X+7);:GOSUB840:POKEX +7,EI
```

```
440 PR=PEEK(X+1)
450 POKE49152+83,H:SYS 49152+82
460 PRINT"  ECHTE  PRUEFSUMME  ";PEEK(X+1)
470 PRINT"  EIGENE PRUEFSUMME  ";PR;:GOSUB840:POKEX+1,EI
480 PRINTCHR$(17)CHR$(17);"  ";CHR$(18);"G";CHR$(146);"LEICHER   HEADER"
490 PRINT"  ";CHR$(18);"A";CHR$(146);"NDRER     HEADER"
500 PRINT"  ";CHR$(18);"T";CHR$(146);"AUSCHE    HEADER"
510 PRINT"  ";CHR$(18);"K";CHR$(146);"OPIERE    HEADER"
520 PRINT"  ";CHR$(18);"E";CHR$(146);"NDE, FORMATIEREN"
530 GET A$:IF A$=""THEN530
540 IF A$="A" THEN GOTO320
550 IF A$="G" THEN PRINTCHR$(147);:GOTO360
560 IF A$="E"THEN630
570 IF A$="T"THEN1020
580 IF A$="K"THEN880
590 GOTO530
600 IF H>SETHENH=0
610 IF H<0 THENH=SE
620 GOTO330
630 PRINTCHR$(147)
640 FORB=1 TO 5:PRINTCHR$(17):NEXT
645 PRINT" GEBEN SIE DEN SPEED DES TRACKS AN  ";SP;:PO=35:GOSUB840
650 IF EI <0 OR EI>3 THEN640
660 POKE49277,EI*32:PRINTCHR$(147);
670 FORN=0TO SE*8-1:PRINTCHR$(19);"    ";CHR$19);SE*8-1-N
680 PRINT#1,"M-W"CHR$(N)CHR$(3)CHR$(1)CHR$(PEEK(49664+N)) :NEXT
690 FORN=0TO 48:PRINTCHR$(19);"    ";CHR$19);48-N
700 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(PEEK(49248+N)) :NEXT
710 PRINT#1,"M-E"CHR$(36)CHR$(4)
720 PRINT#1,"M-R"CHR$(1)CHR$(0)CHR$(1)
730 GET#1,A$:IF ASC (A$+CHR$(0))>128THEN720
740 IF ASC (A$+CHR$(0)) >1THENPRINT"FORMAT ERROR":GOTO800
750 PRINT"FORMAT OK     "
760 PRINT:PRINT:PRINT "  WEITERE TRACKS    "
770 GET A$:IF A$="" THEN770
780 IF A$<>"N" THEN CLOSE1:GOTO50
790 CLOSE1:END
800 PRINT:PRINT:PRINT"  WEITERER VERSUCH"
810 GET A$:IF A$="" THEN810
820 IF A$<>"N" THEN PRINTCHR$(147):GOTO670
```

```
830 GOTO760
840 POKE211,PO:GOTO850
850 SYS58732:INPUT EI
860 IF EI <0 OR EI>255 THEN840
870 RETURN
880 PRINTCHR$(147)CHR$(17)CHR$(17);
885 PRINT" AUF ";CHR$(18);"E";CHR$(146);"INE ODER ";
887 PRINTCHR$(18);"A";CHR$(146);"LLE POSITIONEN KOPIEREN"
890 GET A$:IF A$=""THEN890
900 IF A$="E" THEN  970
910 IF A$ <>"A"THEN 890
920 FORN=0TO SE-1:Y=49664+N*8
930 FORM=0TO 7
940 POKEY+M,PEEK(X+M):NEXT
950 NEXT
960 GOTO480
970 PRINT:PRINT:PRINT" AUF WELCHE POSITION 0 -";SE-1;
980 PO=30:GOSUB840:IF EI>SE-1 THEN PRINT CHR$(147);:GOTO980
990 Y=49664+EI*8
1000 FORM=0TO 7
1010 POKEY+M,PEEK(X+M):NEXT:GOTO480
1020 PRINTCHR$(147)CHR$(17)CHR$(17)" MIT WELCHER POSITION TAUSCHEN 0 -";
SE-1
1030 PRINT:PRINT:PO=17:GOSUB840
1040 IF EI>SE-1 THENPRINTCHR$(147);:GOTO1030
1050 Y=49664+EI*8
1060 FORN=0 TO 7
1070 HE=PEEK(Y+N):POKEY+N,PEEK(X+N)
1080 POKEX+N,HE
1090 NEXT:GOTO480
1100 REM
1110 DATA169,8,153,0,194,200,200,165,247,153,0,194,200,165,248,153,0,194
,200
1020 DATA165,250,153,0,194,200,165,249,153,0,194,200,169,15,153,0,194,20
0
1130 DATA153,0,194,200,169,0,89,250,193,89,251,193,89,252,193,89,253,193
,153
1140 DATA249,193,96,0,0,0,0,0,169,0,133,247,160,0,32,0,192,230,247,165,2
47
```

```
1150 DATA197,251,144,245,96,169,0,10,10,10,24,105,8,168,76,41,192,0,0,16
0
1160 DATA136,162,0,169,35,133,81,169,17,133,67,141,40,6,169,11,141,38,6,
141
1170 DATA32,6,173,0,28,41,159,9,0,141,0,28,76,132,252,169,35,133,8,169,2
24
1180 DATA133,1,165,1,48,252,96,-1
```

Anleitung zum Programm:

Wenn Sie das Programm starten, müssen Sie ein wenig Geduld aufbringen, denn die Daten in den DATA-Zeilen müssen erst in den Speicher gepoket werden. Daraufhin wird die eingelegte Diskette initialisiert und die entsprechende ID angegeben. Es besteht die Möglichkeit, die ID zu ändern. Wenn Sie keine Blockheader-Manipulation durchführen wollen, somit die entsprechende Frage verneinen, wird der angegebene Track fehlerlos formatiert. Zuvor müssen Sie jedoch noch den Speed des Tracks angeben. Soll er nicht vom normalen abweichen, brauchen Sie nur RETURN zu drücken.

Änderung der Header-Parameter:

Als erstes werden Sie nach der Position des zu ändernden Headers gefragt. Die einzige Angabe, die einer Erklärung bedarf, ist die Angabe der Prüfsumme. Die unter 'echte Prüfsumme' ausgegebene Zahl ist die aus den zuvor gemachten Angaben errechnete Prüfsumme. Unter 'eigene Prüfsumme' wird diejenige angezeigt, die man selbst gewählt hat. Im Anschluß an diese Eingaben erscheint ein Menü, bei dem Sie unter folgenden Punkten auswählen können:

1. Gleicher Header
   erlaubt eine erneute Modifikation desselben Headers.

2. Anderer Header
   ermöglicht die Änderung eines weiteren Headers.

3. Tausche Header
   ermöglicht die Vertauschung von zwei Headern und somit eine Änderung der Reihenfolge der Sektoren auf einem Track, was auch einen recht guten Kopierschutz abgibt.

4. Kopiere Header
   erlaubt, einen Header zu duplizieren und dadurch zwei gleiche Sektoren auf einem Track zu erzeugen, was nicht von allen Kopierprogrammen korrekt kopiert werden kann. Rufen Sie diesen Menüpunkt auf, werden Sie gefragt, ob Sie Ihren zuvor erstellten Header auf einen weiteren oder alle anderen Positionen kopieren wollen. Kopieren Sie Ihren Header auf alle anderen Positionen, erhalten Sie einen Track, bei dem alle Sektoren gleich sind. Ein so veränderter Track kann nur sehr schwer kopiert werden. Doch darauf werden wir noch zu einem späteren Zeitpunkt eingehen.

5. Ende, formatieren
   Formatiert den angegebenen Track. Eine Änderung der Speed-Flags ist möglich.

Das Programm besteht aus drei Teilen. Der eine ist der BASIC-Teil, über den alle Eingaben sowie die Vertauschung der Header vorgenommen werden. Der zweite ist ein Maschinenspracheteil, der die Blockheader, die später auf Disk geschrieben werden, im Computer erstellt. Er entspricht in etwa der Routine, die sich auch in der Formatroutine der Floppy befindet. Der dritte Teil ist ein Maschinenspracheprogramm, das in der Floppy gestartet wird. Es übergibt die Parameter, wie beispielsweise die Länge der Lücke zwischen den Sektoren, an die Formatroutine.

Wie schon gesagt, werden die Blockheader erst im Computer erzeugt, um nach ihrer Änderung in die Floppy ab Adresse $0300 geschickt zu werden, wo sie in das GCR-Format gewandelt und auf Diskette aufgetragen werden.

Im folgenden zeigen wir Ihnen die zwei erwähnten Maschinenspracheteile. Als erstes den Teil, der im Computer abläuft und die Blockheader im Speicher des Rechners erzeugt.

Das Programm, das die Daten für die Blockheader erzeugt, wird ab $C040 (49216) gestartet und legt sie ab $C200 (49664) im Speicher ab.

```
C000 LDA #$08     $08 laden (Zeichen für Blockheader)
C002 STA $C200,Y und übergeben
C005 INY          Zeiger erhöhen
C006 INY          erhöhen, um Lücke für Prüfsumme zu lassen
C007 LDA $F7      Sektornummer holen
C009 STA $C200,Y und speichern
C00C INY          Zeiger erhöhen
C00D LDA $F8      Track-Nummer holen
C00F STA $C200,Y und speichern
C012 INY          Zeiger erhöhen
C013 LDA $FA      zweite ID holen
C015 STA $C200,Y und übergeben
C018 INY          Zeiger erhöhen
C019 LDA $F9      erste ID holen
C01B STA $C200,Y und übergeben
C01E INY          Zeiger erhöhen
C01F LDA #$0F     $0F (Lückenbyte) holen
C021 STA $C200,Y und speichern
C024 INY          Zeiger erhöhen
C025 STA $C200,Y und Lückenbyte erneut speichern
C028 INY          Zeiger erhöhen
C029 LDA #$00     Akku mit $00 laden (normalisieren)
C02B EOR $C1FA,Y
C02E EOR $C1FB,Y Prüfsumme über den
C031 EOR $C1FC,Y Blockheader berechnen
C034 EOR $C1FD,Y
C037 STA $C1F9,Y und Prüfsumme abspeichern
C03A RTS          Rücksprung
```

Für die Erstellung der Blockheader wird des Programm hier gestartet.

```
C040 LDA #$00     Sektornummer auf 0 stellen
C042 STA $F7
C044 LDY #$00     Zeiger auf 0 setzen
C046 JSR $C000    und Blockheader erstellen
C049 INC $F7      Sektornummer erhöhen
C04B LDA $F7      Nummer laden
C04D CMP $FB      und mit maximaler Zahl vergleichen.
C04F BCC $C046    verzweige, wenn Maximum nicht erreicht
C051 RTS          Rücksprung
```

Für die Errechnung der Prüfsumme eines einzelnen Blockheaders wird die Routine hier gestartet.

```
C052 LDA #$00     Position des Headers laden (variabel)
C054 ASL
C055 ASL          Mit 8 multiplizieren
C056 ASL
C057 CLC          Carry für Addition löschen
C058 ADC #$08     und mit 8 addieren,
C05A TAY          als Zählwert übergeben
C05B JMP $C029    Prüfsumme errechnen
```

Die folgende Routine wird in der Floppy ab $0400 gestartet, nachdem die Blockheader ebenfalls in die Floppy gesandt wurden. Alle nötigen Parameter werden mit der zu startenden Routine mitgesandt. Sie wird ab $0424 gestartet.

```
0400 LDY #$88     LOW-Byte der Endadresse der Blockheader
0402 LDX #$00     X-Register löschen
0404 LDA #$23     Zeiger auf Ende der Formatierung setzen
0406 STA $51      damit nur ein Track formatiert wird
0408 LDA #$11     Maximale Sektoranzahl laden (variabel)
040A STA $43      und übergeben
040C STA $0628
```

```
040F LDA #$0B     Lücke zwischen den Sektoren laden
0411 STA $0626    und übergeben
0414 STA $0620    gleichzeitig als Zähler für Fehler nehmen
0417 LDA $1C00    Control-Port laden
041A AND #$9F     Speed-Flags löschen
041C ORA #$00     Mit eigenem Speed verknüpfen (hier 00)
041E STA $1C00    und speichern
0421 JMP $FC84    Formatierung beginnen
```

Das Programm wird hier gestartet

```
0424 LDA #$23     Nummer des zu formatierenden Tracks
0426 STA $08      übergeben
0428 LDA #$E0     Job-Code 'E0' für Programm starten
042A STA $01      in Job-Speicher schreiben
042C LDA $01      Code für Rückmeldung empfangen?
042E BMI $042C    verzweige, wenn nicht empfangen
0430 RTS          Rücksprung
```

Nachdem wir nun das Programm erklärt haben, wollen wir uns nun mit seiner Anwendung beschäftigen.

6.2.5 Das Arbeiten mit zerstörten Blöcken

Wie wir schon sagten, kann man mit dem obigen Programm die Blockheader-Parameter ändern und dadurch Sektoren gezielt zerstören. Dieses Verfahren dient, bis auf einige Sonderfälle, kaum zum Schutz gegen das Kopieren, sondern nur zur Verhinderung des Zugriffs anderer auf einen so geschützten Block. Jetzt stellt sich jedoch die Frage, wie man überhaupt Daten auf diesen Blöcken speichern oder wieder lesen soll, denn die Floppy fängt jedesmal an zu blinken, wenn man versucht, den Block auf die herkömmliche Weise zu laden.

Sie können sich sicher denken, daß es natürlich einen Weg gibt, diese Blöcke zu nutzen. Die einfachste Methode, die sich auch für alle Arten von zerstörten Blockheadern einsetzen läßt, ist

die: Man liest den Blockheader vor dem zerstörten Header ein. Die darauffolgende SYNC-Markierung ist folglich die des Datenblocks des gelesenen Headers. Wir überspringen mit unserem Floppy-Programm einfach diese Markierung. Die nächste SYNC-Markierung, die gefunden wird, ist dann die unseres zerstörten Blockheaders. Diese Markierung wird ebenfalls nicht beachtet. Die daraufhin gefundene SYNC-Markierung muß dann die des Datenblocks sein, der zu dem zerstörten Header gehört. Nachdem diese Markierung gefunden wurde, wird der Datenblock gelesen. Sie sehen also, daß es uns mit diesem Trick möglich ist, Daten von einem auch noch so zerstörten Block zu lesen, was uns sonst nicht möglich gewesen wäre. Nach dem gleichen System können wir auch Daten speichern. Sollten Sie mehrere Blockheader hintereinander zerstört haben, so brauchen Sie nur entsprechend viele SYNC-Markierungen zu überlesen.

Mit dem folgenden Floppy-Programm ist es möglich, eine beliebige Anzahl von Sektoren zu überspringen und dann den gewünschten Block zu lesen. Das Programm wird ab $0522 gestartet.

```
0500 JSR $F50A   Blockheader suchen, auf Datenblock warten
0503 LDX #$01    Zähler für Sektor überspringen (variabel)
0505 JSR $0512   Routine zum Überspringen eines Sektors
0508 DEX         Zähler verringern
0509 BNE $0505   verzweige, wenn nicht abgelaufen
050B LDA #$06    HIGH-Byte für Pufferzeiger auf $06 stellen
050D STA $31     damit die Daten ab $0600 geladen werden
050F JMP $F4D4   in Routine für Block laden springen

0512 LDA $1C00   Control-Port laden
0515 BPL $0512   verzweige, wenn SYNC noch anliegt
0517 JSR $F556   auf neue SYNC-Markierung warten
051A LDA $1C00   Control-Port laden
051D BPL $051A   verzweige, wenn SYNC noch anliegt
051F JMP $F556   auf neue SYNC-Markierung warten, RTS
```

Das Programm wird hier gestartet

```
0522 LDA #$01    Track auf dem geladen wird (variabel)
0524 STA $0A     übergeben
0526 LDA #$00    Sektor, der geladen wird (variabel)
0528 STA $0B     übergeben
052A LDA #$E0    Job-Code für Programm ausführen, laden
052C STA $02     und übergeben
052E LDA $02     Rückmeldung abwarten
0530 BMI $052E   verzweige, wenn keine Rückmeldung
0532 RTS         Rücksprung
```

Im Anschluß daran das gewohnte BASIC-Listing. Es erlaubt, die Eingabe des zu lesenden Tracks, des Sektor Sektors sowie die Eingabe der zu überspringenden Sektoren. Die Nummer des tatsächlich gelesenen Sektors, ergibt sich aus der Nummer des eingegebenen Sektors, addiert mit der Anzahl der zu überspringenden Sektoren. Die Daten des zu lesenden Blocks werden ab $0600 (Puffer 3) in der Floppy abgelegt.

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 5477 THEN PRINT"ERROR IN DATAS":STOP
105 INPUT "WELCHER TRACK";T
110 INPUT "WELCHER SEKTOR";S
115 INPUT "ZU UEBERLESENE SEKTOREN";U
120 PRINT#1,"M-W"CHR$(35)CHR$(5)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(39)CHR$(5)CHR$(1)CHR$(S)
130 PRINT#1,"M-W"CHR$(4)CHR$(5)CHR$(1)CHR$(U)
135 PRINT#1,"M-E"CHR$(34)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(2)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A>127 THEN130
170 IFA =1 THENPRINT"OK":END
```

```
180 PRINT"ERROR":END
302 DATA 32, 10,245,162,  1, 32, 18,  5,202,208,250,169,  6,133, 49, 76,
212,244
304 DATA173,  0, 28, 16,251, 32, 86,245,173,  0, 28, 16,251, 76, 86,245,
169,  1
306 DATA133, 10,169,  0,133, 11,169,224,133,  2,165,  2, 48,252, 96, -1
```

Wie das Lesen eines Blocks, so erfolgt auch das Schreiben. Das hierzu benötigte Programm sieht jedoch etwas anders aus, da die SAVE-Routine der Floppy später angesprungen werden muß. Die von ihr abgearbeiteten Programmteile müssen zum Teil in das eigene Programm übernommen werden.

Hier nun die entsprechende SAVE-Routine, die ab $0400 in der Floppy gestartet wird.

```
0500 LDA #$06    HIGH-Byte des Puffers auf $06 stellen,
0502 STA $31     um die Daten von $0600 zu speichern
0504 JSR $F5E9   Prüfsumme über Datenblock berechnen
0507 STA $3A     und abspeichern
0509 LDA $1C00   Control-Port laden
050C AND #$10    und auf Schreibschutz prüfen
050E BNE $0515   verzweige, wenn kein Schreibschutz
0510 LDA #$08    Fehlermeldung im Akku
0512 JMP $F969   und zur Ausgabe springen
0515 JSR $F78F   Pufferinhalt ins GCR-Format wandeln
0518 JSR $F510   Blockheader suchen
051B LDX #$01    Zähler für Sektor überspringen (variabel)
051D JSR $0526   Routine zum Überspringen eines Sektors
0520 DEX         Zähler verringern
0521 BNE $051D   verzweige, wenn nicht abgelaufen
0523 JMP $F58C   Block auf Diskette schreiben

0526 LDA $1C00   Control-Port laden
0529 BPL $0526   verzweige, wenn SYNC-Signal noch anliegt
052B JSR $F556   auf nächste SYNC-Markierung warten
```

```
052E LDA $1C00   Control-Port lesen
0531 BPL $052E   verzweige, wenn SYNC-Signal noch anliegt
0533 JMP $F556   auf nächste SYNC-Markierung warten, RTS
```

Das Programm wird hier gestartet

```
0536 LDA #$01    Track, auf dem geladen wird (variabel)
0538 STA $0A     übergeben
053A LDA #$00    Sektor, der geladen wird (variabel)
053C STA $0B     übergeben
053E LDA #$E0    Job-Code für Programm ausführen, laden
0540 STA $02     und übergeben
0542 LDA $02     Rückmeldung abwarten
0544 BMI $0542   verzweige, wenn keine Rückmeldung
0546 RTS         Rücksprung
```

Jetzt folgt wieder der BASIC-Loader, bei dem Sie Track und Sektor sowie die Anzahl der zu überspringenden Blöcke eingeben können.

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 7633 THEN PRINT"ERROR IN DATAS":STOP
105 INPUT "WELCHER TRACK";T
110 INPUT "WELCHER SEKTOR";S
115 INPUT "ZU UEBERLESENE SEKTOREN";U
120 PRINT#1,"M-W"CHR$(55)CHR$(5)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(59)CHR$(5)CHR$(1)CHR$(S)
130 PRINT#1,"M-W"CHR$(28)CHR$(5)CHR$(1)CHR$(U)
135 PRINT#1,"M-E"CHR$(54)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(2)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A>127 THEN130
```

```
170 IFA =1 THENPRINT"OK":END
180 PRINT"ERROR":END
302 DATA169,  6,133, 49, 32,233,245,133, 58,173,  0, 28, 41, 16,208,  5,
169,  8
304 DATA 76,105,249, 32,143,247, 32, 16,245,162,  1, 32, 38,  5,202,208,
250, 76
306 DATA140,245,173,  0, 28, 16,251, 32, 86,245,173,  0, 28, 16,251, 76,
 86,245
308 DATA169,  1,133, 10,169,  0,133, 11,169,224,133,  2,165,  2, 48,252,
 96, -1
```

Nachdem wir gesehen haben, wie wir uns mit Hilfe von Blockheader-Manipulationen vor dem Zugriff Fremder auf unsere Blöcke schützen können, wollen wir uns einmal ansehen, wie man mit dieser Methode einen sehr guten Kopierschutz aufbauen kann. Bei der Änderung von mehreren Blockheadern auf dem gleichen Track kann es zu Störungen beim Einlesen eines Blocks kommen.

6.2.6 Gleiche Blöcke auf einem Track

Eine sehr sichere Kopierschutzmethode ist es, einen Track nur mit gleichen Blockheader Blockheadern zu beschreiben. Einen solchen Track kann man mit dem schon beschriebenen Formatierungsprogramm mühelos erstellen.

Wie arbeitet ein solcher Kopierschutz und warum ist er so schwer, fast unmöglich, zu kopieren? Um dies zu verstehen, müssen wir uns etwas näher mit dem System der Kopierprogramme beschäftigen.

Das Kopierprogramm lädt einen oder mehrere Blöcke zugleich in den Speicher der Floppy und schickt daraufhin die Daten zum Computer, da der Speicher der Floppy nicht ausreicht, weitere Daten zu speichern. Durch das Senden der Daten an den Computer muß das Kopierprogramm die Stelle wiederfinden, an der es aufgehört hat, Daten zu lesen. Es sucht wieder die Anfangsstelle und überliest eine entsprechende Anzahl von Daten, um daraufhin die neuen Daten einzulesen. Das Programm erkennt

einen schon vollständig eingelesenen Track daran, daß es dieselben Daten findet, die schon zu Beginn eingelesen wurden. Wenn Sie sich jetzt daran erinnern, wie unser modifizierter Track aussieht, werden Sie feststellen, daß das Kopierpogramm keine Möglichkeit hat, sich auf dem Track zu orientieren, da alle Blöcke gleich aussehen. Für das Programm ist es somit fast unmöglich, die Anzahl der SYNC-Markierungen sowie die Anzahl der Bytes auf diesem Track festzustellen. Auch die Kopierprogramme, die mit einem parallelen Bus arbeiten, welcher die Daten ohne "abzusetzen" einlesen kann, bekommen Probleme, die Anzahl der SYNC-Markierungen richtig festzustellen.

Dieser Track ist so wirkungsvoll, weil es für Kopierprogramme sehr schwer ist, ihn richtig zu analysieren.

Wir wissen also, womit die Kopierprogramme zu "kämpfen" haben, doch stellt sich hier die Frage, wie wir selbst einen solchen Track auf seine Richtigkeit überprüfen.

Diese Überprüfung ist recht aufwendig, da sie in drei Etappen durchgeführt wird. Als erstes wird eine große Anzahl von Bytes eingelesen und gleichzeitig die Anzahl der SYNC-Markierungen gezählt. Diese Zahl ist, falls der Track korrekt ist, recht konstant. Als nächstes wird überprüft, wie lang die SYNC-Markierungen selber sind, damit das Kopierprogramm nicht, wenn es nicht genügend Daten feststellt, die entstehende Lücke mit längeren SYNC-Markierungen vertuschen kann. Als drittes und letztes muß noch geprüft werden, ob die Blockheader des entsprechenden Tracks auch wirklich alle gleich sind. Sollten alle Prüfungen positiv ausfallen, so liegt ein "Original" vor, was durch eine entsprechende Rückmeldung von der Kopierschutzabfrage signalisiert wird. Ein Programm, das all diese Prüfdurchgänge enthält, werden Sie jetzt anschließend sehen. Es liegt (in der Floppy) ab Adresse $0400 im Speicher und wird ab $04DF gestartet. Die entsprechende Rückmeldung an den Rechner wird in $10 der Floppy gespeichert. 0 steht für "Kopierschutz nicht richtig erkannt" und $FF für "alles OK".

```
0400 LDA $08       Nummer des zu prüfenden Tracks laden
0402 LDX #$04      Zähler für die vier Zonenabschnitte
0404 CMP $04DA,X   Nummer mit Zahl aus Tabelle vergleichen
0407 DEX           Zähler verringern
0408 BCS $0404     verzweige, wenn Nummer größer gleich der
                   Zahl aus der Tabelle
040A TXA           entsprechenden Zählerwert in den Akku
040B ASL           schieben und dann die Bits an Position
040C ASL           5 und 6 schieben, um sie als Maßzahl
040D ASL           für den Speed auf dem entsprechenden
040E ASL           Track zu benutzen
040F ASL
0410 STA $44       Wert zwischenspeichern
0412 LDA $1C00     Control-Port laden
0415 AND #$9F      Bits für Speed löschen
0417 ORA $44       mit eigenem Speed verknüpfen
0419 STA $1C00     und Wert übergeben
041C LDX #$00      LOW und HIGH-Byte der Anzahl der zu
041E LDY #$1F      lesenden Bytes laden (7936)
0420 LDA #$00      Zählwert für SYNCs auf
0422 STA $37       Null setzen
0424 LDA $1C00     Control-Port laden
0427 BPL $0424     verzweige, wenn SYNC noch anliegt
0429 CLV           Byte-Ready-Leitung löschen
042A BVC $042A     warten, bis Byte anliegt
042C CLV           Byte-Ready-Leitung wieder löschen
042D DEX           Byte-Zähler verringern
042E BNE $0433     verzweige, wenn kein Übertrag
0430 DEY           sonst auch HIGH-Byte verringern
0431 BEQ $043D     verzweige, wenn Zähler abgelaufen
0433 LDA $1C00     Control-Port laden
0436 BMI $0429     verzweige, wenn kein SYNC anliegt
0438 INC $37       sonst SYNC-Zähler erhöhen
043A JMP $0424     und auf neue SYNC-Markierung warten
043D LDX $37       Zähler für gefundene SYNC-Markierungen
043F STX $04FF     in $04FF speichern
```

Der jetzt folgende Programmteil prüft die Länge der SYNC-Markierungen. Zu diesem Zweck wird der Timer 1 der Ein/Ausgabe VIAs benutzt.

```
0442 LDA #$C0     LOW-Byte des Zählwerts für SYNC Länge
0444 STA $1804    ins LOW-Byte des Timers schreiben
0447 LDA $1C00    Control-Port laden und auf Ende
044A BPL $0447    der SYNC-Markierung warten
044C LDA $1C00    Control-Port laden und auf Anfang
044F BMI $044C    der SYNC-Markierung warten
0451 LDA #$80     HIGH-Byte des Timerwertes laden und
0453 STA $1805    übergeben, Timer starten
0456 LDA $1C00    Control-Port laden und auf Ende
0459 BPL $0456    der SYNC-Markierung warten
045B LDA $1805    HIGH-Byte des Timerwertes laden
045E BPL $0465    verzweige, wenn Wert kleiner $80 (falsch)
0460 DEX          Zähler für Anzahl der SYNCs verringern
0461 BNE $0442    und nächste Markierung abfragen
0463 LDA #$FF     Rückmeldewert laden
0465 STA $04FE    und in $04FE speichern
```

Als nächstes folgt der Teil, in dem 25 aufeinanderfolgende Blockheader geladen und ab $0500 gespeichert werden. Es gibt zwar im Normalfall keine 25 verschiedenen Blockheader auf einem Track, jedoch geht man so sicher, daß alle geladen werden. Dies Teilprogramm kann auch separat verwendet werden, um die Reihenfolge der Sektoren oder ähnliches zu überprüfen.

```
0468 LDX #$00     Zähler auf Null setzen
046A JSR $F556    Auf SYNC-Markierung warten
046D LDY #$09     Zähler für Anzahl der Bytes pro Header
046F BVC $046F    warten, bis ein Byte anliegt
0471 CLV          Byte-Ready-Leitung löschen
0472 LDA $1C01    Byte vom Port holen
0475 CMP #$52     mit Wert für Blockheader vergleichen
0477 BNE $046A    verzweige, wenn kein Blockheader
0479 STA $0500,X  sonst Wert speichern
```

```
047C INX          Zähler erhöhen
047D NOP
047E BVC $047E    auf neues Byte warten
0480 CLV          Byte-Ready-Leitung löschen
0481 LDA $1C01    Byte vom Port holen
0484 STA $0500,X  und speichern
0487 INX          Zähler erhöhen
0488 BEQ $048F    verzweige, wenn alle Header geholt
048A DEY          Zähler für Bytes pro Header verringern
048B BNE $047E    verzweige, wenn nicht alle Bytes geholt
048D BEQ $046A    unbedingter Sprung
048F LDA #$05     HIGH-Byte des zu bearbeitenden Puffers auf
0491 STA $31      $05 stellen (Puffer 2)
0493 JSR $F8E0    Bytes ins Bin.-Format wandeln
0496 LDX #$FF     Zähler setzen, um die Daten umzukopieren
0498 LDA $04FF,X  Daten holen
049B STA $0500,X  und ein Byte weiter abspeichern
049E DEX          Zähler verringern
049F CPX #$FF     vergleiche, ob schon alles kopiert
04A1 BNE $0498    verzweige, wenn nicht allen kopiert
04A3 LDA $38      erstes Byte, das bei der Umwandlung ins
04A5 STA $0500    Bin-Format gespeichert wurde, zurück-
                  schreiben
```

Der folgende Teil dient zur Überprüfung der gelesenen Blockheader.

```
04A8 LDY #$C8     Zeichenanzahl laden
04AA LDX #$08     Zeichenanzahl pro Header
04AC LDA $04FF,X  Zeichen holen
04AF CMP $04FF,Y  mit erstem Header vergleichen
04B2 BNE $04C6    verzweige, wenn Zeichen ungleich
04B4 DEX          Zähler verringern
04B5 DEY          Zähler verringern
04B6 BEQ $04BE    verzweige, wenn alle Zeichen verglichen
04B8 CPX #$00     kontrollieren, ob ganzer Header verglichen
04BA BNE $04AC    verzweige, wenn nein
04BC BEQ $04AA    unbedingter Sprung
04BE LDA $04FE    Rückmeldung der SYNC-Prüfroutine holen
```

```
04C1 CMP #$FF     auf Richtigkeit überprüfen
04C3 BNE $04D7    verzweige, wenn Fehler
04C5 LDA $04FF    Anzahl der gefundenen Sektoren holen
04C8 CMP #$2A     mit 42 vergleichen
04CA BCC $04D7    Fehler, wenn der Wert kleiner ist
04CC CMP #$2E     Wert mit 46 vergleichen
04CE BCS $04D7    Fehler, wenn Wert größer oder gleich ist
04D0 LDA #$FF     Positive Rückmeldung an Computer übergeben
04D2 STA $10      Rückmeldung schreiben
04D4 JMP $FD9E    Rücksprung
04D7 LDA #$00     negative Rückmeldung laden
04D9 BEQ $04D2    unbedingter Sprung
.
04DB 2B 1F 19 12 Werte für die vier Zonenabschnitte
```

Das Programm wird hier gestartet

```
04DF LDA #$01     Track-Nummer laden
04E1 STA $08      und an Job übergeben
04E3 LDA #$E0     Job-Code 'E0' laden (Programm ausführen)
04E5 STA $01      in Job-Seicher schreiben
04E7 LDA $01      Rückmeldung abwarten
04E9 BMI $04E7    verzweige, wenn noch keine Rückmeldung
04EB RTS          Rücksprung
```

In diese Routine wurde eine eigene Routine eingebaut, die den Speed des jeweiligen Tracks bestimmt, da diese Routine im Betriebssystem der Floppy diese Aufgabe nur bis Track 35 fehlerlos durchführt. Im Anschluß an das Maschinenprogramm folgt wieder unser BASIC-Loader.

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 28329 THEN PRINT"ERROR IN DATAS":STOP
```

```
110 INPUT "WELCHER TRACK";T
120 PRINT#1,"M-W"CHR$(224)CHR$(4)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(223)CHR$(4)
135 FORN=1TO 500:NEXT
140 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
170 IFA =255 THENPRINT"SCHUTZ OK":END
180 PRINT"SCHUTZ ERROR":END
302 DATA165,  8,162,  4,221,218,  4,202,176,250,138, 10, 10, 10, 10, 10,
133, 68
304 DATA173,  0, 28, 41,159,  5, 68,141,  0, 28,162,  0,160, 31,169,  0,
133, 55
306 DATA173,  0, 28, 16,251,184, 80,254,184,202,208,  3,136,240, 10,173,
  0, 28
308 DATA 48,241,230, 55, 76, 36,  4,166, 55,142,255,  4,169,192,141,  4,
 24,173
310 DATA  0, 28, 16,251,173,  0, 28, 48,251,169,128,141,  5, 24,173,  0,
 28, 16
312 DATA251,173,  5, 24, 16,  5,202,208,223,169,255,141, 254,  4,162,  0
, 32, 86
314 DATA245,160,  9, 80,254,184,173,  1, 28,201, 82,208,241,157,  0,  5,
232,234
316 DATA 80,254,184,173,  1, 28,157,  0,  5,232,240,  5,136,208,241,240,
219,169
318 DATA  5,133, 49, 32,224,248,162,255,189,255,  4,157,  0,  5,202,224,
255,208
320 DATA245,165, 56,141,  0,  5,160,200,162,  8,189,255,  4,217,255,  4,
208, 18
322 DATA202,136,240,  6,224,  0,208,240,240,236,173,254,  4,201,255,208,
 18,173
324 DATA255,  4,201, 42,144, 11,201, 46,176,  7,169,255,133, 16, 76,158,
253,169
326 DATA  0,240,247, 43, 31, 25, 18,169,  1,133,  8,169,224,133,  1,165,
  1, 48
328 DATA252, 96, -1
```

Aufbringen können Sie diesen Schutz, indem Sie das Programm zum Ändern der Blockheader-Parameter aus Kapitel 6.2.4 ver-

wenden und dort, wie in der Anleitung schon erwähnt, "kopiere Header" anwählen, um daraufhin den entsprechenden Header auf *alle* anderen Positionen zu kopieren.

6.3 Halftracks

Was sind überhaupt Halftracks, und wie kann man sie für einen Kopierschutz einsetzen? Diese Fragen sollen in dem folgenden Kapitel erörtert werden.

Um das Arbeiten mit Halftracks (Halbspuren) zu verstehen, müssen Sie wissen, daß sich der R/W-Kopf der Floppy nicht nur in der von uns bekannten Schrittweite von einer Spur, sondern in halben Spuren bewegt. Diese halben Spuren sind für die Floppy problemlos zu lesen, das Schreiben hingegen funktioniert nicht so gut. Dieser Umstand ist leicht zu erklären. Das nackte Laufwerk der Diskettenstation 1541 ist eigentlich für 80 Spuren ausgelegt. Commodore nutzt es jedoch nur als 40 Spur-Laufwerk, weshalb man auch einen 40 Spur-Schreibkopf verwendet hat. Den ursprünglichen Lesekopf, der für 80 Spuren gedacht war, hat man jedoch beibehalten. Beim Beschreiben einer Halbspur überschreibt man wegen des breiteren Schreibkopfes daher auch einen Teil der benachbarten Spuren, auf denen man Daten zerstört. Es ist also nicht möglich, drei nebeneinanderliegende Halbspuren mit der 1541 zu schreiben.

Diesen Umstand haben sich einige Softwarehäuser zunutze gemacht und haben auf ihren Disketten mit Hilfe eines anderen Laufwerks drei Halbspuren aufgetragen, die mit der 1541 zwar gelesen und somit auf ihre Richtigkeit überprüft, jedoch nicht reproduziert werden können. Sie werden einsehen, daß es sich bei diesem Verfahren um den absolut sicheren Kopierschutz handelt.

Wie haben allerdings einen Weg gefunden, wie man solche Halbspuren auch mit der 1541 auftragen und wieder abfragen kann. Man kann mit Gewißheit sagen, daß dieser Kopierschutz zu den sichersten gehört, die mit der 1541 aufgetragen werden können. Möglicherweise ist er sogar der sicherste.

Das Auftragen des Kopierschutzes geschieht folgendermaßen: Zuerst werden die drei benachbarten Halbspuren von allen SYNC-Markierungen befreit. Dann wird auf der ersten Halbspur eine SYNC-Markierung geschrieben, worauf einige wenige Daten folgen. Danach wird der R/W-Kopf schnell auf die benachbarte Halbspur gefahren und ebenfalls eine SYNC-Markierung mit Daten aufgetragen. Derselbe Vorgang vollzieht sich noch ein drittes Mal. Dieses dreimalige Schreiben von Daten und das Verschieben des Kopfes erfolgt so schnell, daß sich die Diskette wärend dieser Operation nicht vollständig drehen kann. Es werden beim Schreiben der Halbspuren die benachbarten Halbspuren überschrieben, jedoch nur an Stellen, bei denen auf den benachbarten Spuren keine Daten liegen. Dies ist nur zu realisieren, wenn der gesamte Schreibprozeß, wie in unserem Fall, weniger Zeit beansprucht als die Zeit, in der sich die Diskette einmal dreht.

Das Kopierprogramm liest immer einen gesamten Track ein und schreibt diesen auch wieder als ganzen Track. Durch dieses Vorgehen werden alle Daten der benachbarten Halbspuren zwangsläufig zerstört. Selbst wenn das Kopierprogramm "wissen" sollte, um welchen Schutz es sich handelt, kann es noch nicht wissen, an welchen Stellen es die Halbspur innerhalb einer Umdrehung wechseln soll.

Sie sehen sicher ein, daß es sich auch bei diesem mit der 1541 aufgetragenen Verfahren um einen absolut sicheren Kopierschutz handelt.

Folgend zeigen wir Ihnen das Programm, mit dem Sie selbst den eben erklärten Kopierschutz auftragen können. Das Programm wird in der Floppy ab $0503 gestartet.

```
0500 JMP $0519   Überspringen des Hauptprogramms
```

Das Programm wird hier gestartet

```
0503 JSR $D042   Disk initialisieren
0506 LDA #$27    Track-Nummer (39)
0508 STA $0A     an Job übergeben
050A LDA #$00    Sektor-Nummer laden
050C STA $0B     und übergeben (nicht nötig)
050E LDA #$E0    Job-Code 'E0' laden
0510 STA $02     und übergeben
0512 BIT $02     auf Rückmeldung warten
0514 BMI $0512   verzweige, wenn noch keine Rückmeldung
0516 JMP $D042   Disk initialisieren, Rücksprung

0519 LDA $1C00   Control-Port laden
051C AND #$9F    Speed auf 00 setzen und
051E ORA #$08    LED aum Laufwerk anschalten
0520 STA $1C00   Wert speichern
0523 JSR $FE0E   Track mit $55 löschen (Löschen der SYNCs)
0526 JSR $FE00   wieder auf Lesen umschalten
0529 LDA #$02    Zähler für die Anzahl der Halbspuren
052B STA $3B     setzen
052D JSR $05A0   Kopf un eine Halbspur verschieben
0530 JSR $FE0E   Track mit #55 löschen
0533 JSR $FE00   Kopf wieder auf Lesen umschalten
0536 DEC $3B     Zähler verringern
0538 BNE $052D   nächste Halbspur löschen
053A LDA $1C0C
053D AND #$1F
053F ORA #$C0    Kopf auf Schreiben
0541 STA $1C0C   umschalten und Port auf Ausgang schalten
0544 LDA #$FF
0546 STA $1C03
0549 STA $1C01   $FF zum Kopf schicken um SYNC zu schreiben
054C LDX #$C8    Zähler für Länge der SYNC-Markierung
054E BVC $054E   warten, bis Byte geschrieben
0550 CLV         Byte-Ready-Leitung löschen
0551 DEX         Zähler verringern
0552 BNE $054E   verzweige, wenn nicht fertig geschrieben
0554 LDA $0580,X Daten zum Schreiben holen
0557 BVC $0557   und warten, bis Byte geschrieben
```

```
0559 CLV          Byte-Ready-Leitung löschen
055A STA $1C01    Byte zu R/W-Kopf schicken
055D INX          Zähler für Byte-Anzahl erhöhen
055E CPX #$08     schon alle Bytes geschrieben
0560 BNE $0554    verzweige, wenn noch Bytes zu schreiben
0562 BVC $0562    warten, bis letztes Byte geschrieben
0564 CLV          Byte-Ready-Leitung löschen
0565 JSR $FE00    Kopf auf Lesen stellen
0568 JSR $05A3    R/W-Kopf um eine Halbspur zurückschieben
056B LDA $0555    LOW-Byte der Adresse, von der
056E CLC          die zu schreibenden Daten geholt
056F ADC #$08     werden, um 8 erhöhen
0571 STA $0555    und Wert wieder speichern
0574 CMP #$98     vergleichen, ob alle Daten geschrieben
0576 BNE $053A    verzweige, wenn nein
0578 JSR $05A0    R/W-Kopf auf Anfangsposition stellen
057B JMP $FD9E    und Rücksprung
057E BRK
057F BRK

0580 69 A5 5A 96 56 59 A6 A9
0588 59 9A 6A 65 66 55 99 AA  Die zu schreibenden
0590 66 95 96 69 A5 5A 6A 6A  Daten
0598 00 00 00 00 00 00 00 00

05A0 LDA #$CA     Wert für Befehl 'DEX' laden
05A2 Byte $2C     Skip nach $05A5
05A3 LDA #$E8     Wert für Befehl 'INX' laden
05A5 STA $05AB    und Wert nach $05AB schreiben
05A8 LDX $1C00
05AB DEX
05AC TXA
05AD AND #$03
05AF STA $4B      R/W-Kopf um eine Halbspur
05B1 LDA $1C00    nach innen oder außen fahren
05B4 AND #$FC
05B6 ORA $4B
05B8 STA $1C00
05BB LDX #$10
05BD LDY #$00
```

```
05BF DEY          Zeitschleife, um dem R/W-Kopf
05C0 BNE $05BF    Zeit zu geben, sich zu positionieren
05C2 DEX
05C3 BNE $05BF
05C5 RTS          Rücksprung
```

Im Anschluß an das Assemblerlisting wieder der BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 20644 THEN PRINT"ERROR IN DATAS":STOP
130 PRINT#1,"M-E"CHR$(3)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(2)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A >127 THEN 145
170 IFA =1 THENPRINT"OK":END
180 PRINT"ERROR":END
302 DATA 76, 25,  5, 32, 66,208,169, 39,133, 10,169,  0,133, 11,169,224,
133,  2
304 DATA 36,  2, 48,252, 76, 66,208,173,  0, 28, 41,159,  9,  8,141,  0,
 28, 32
306 DATA 14,254, 32,  0,254,169,  2,133, 59, 32,160,  5, 32, 14,254, 32,
  0,254
308 DATA198, 59,208,243,173, 12, 28, 41, 31,  9,192,141, 12, 28,169,255,
141,  3
310 DATA 28,141,  1, 28,162,200, 80,254,184,202,208,250,189,128,  5, 80,
254,184
312 DATA141,  1, 28,232,224,  8,208,242, 80,254,184, 32,  0,254, 32,163,
  5,173
314 DATA 85,  5, 24,105,  8,141, 85,  5,201,152,208,194, 32,160,  5, 76,
158,253
316 DATA 40,  0,105,165, 90,150, 86, 89,166,169, 89,154,106,101,102, 85,
153,170
318 DATA102,149,150,105,165, 90,106,106,  0,  0,  0,  0,  0,  0,  0,  0,
169,202
```

```
320 DATA 44,169,232,141,171,  5,174,  0, 28,202,138, 41,  3,133, 75,173,
  0, 28
322 DATA 41,252,  5, 75,141,  0, 28,162, 16,160,  0,136,208,253,202,208,
250, 96
324 DATA -1
```

Wie Sie auch schon anhand des dokumentierten Listings sehen konnten, wird der Schutz auf den Spuren oberhalb von 35 aufgetragen. Doch kommen wir jetzt zu dem Abfrageprogramm, das auch ab $0503 gestartet wird.

```
0500 JMP $0519   Überspringen des Hauptprogramms
```

Das Programm wird hier gestartet:

```
0503 JSR $D042   Disk initialisieren
0506 LDA #$27    Track-Nummer (39)
0508 STA $0A     an Job übergeben
050A LDA #$00    Sektor-Nummer laden
050C STA $0B     und übergeben (nicht nötig)
050E LDA #$E0    Job-Code 'E0' laden
0510 STA $02     und übergeben
0512 LDA $02     auf Rückmeldung warten
0514 BMI $0512   verzweige, wenn noch keine Rückmeldung
0516 JMP $D042   Disk initialisieren, Rücksprung

0519 LDA $1C00   Control-Port laden
051C AND #$9F    Speed auf 0 stellen und
051E ORA #$08    LED anschalten
0520 STA $1C00   Wert speichern
0523 NOP
0524 NOP
0525 JSR $0593   Zähler für SYNC auf 0 setzen
0528 NOP
0529 LDA #$00    Zähler für Fehlversuche beim Finden der
052B STA $3B     richtigen Anfangsstelle auf 0 setzen
052D LDA $1C00   Control-Port laden
0530 BPL $052D   warten, bis keine SYNC-Markierung anliegt
```

```
0532 LDA $1C00    Control-Port laden
0535 BPL $0544    verzweige, wenn SYNC-Signal anliegt
0537 INC $05FE    sonst Zähler erhöhen
053A BNE $0532    verzweige, wenn kein Überlauf
053C INC $05FF    HIGH-Byte des Zählers erhöhen
053F BNE $0532    verzweige, wenn kein Überlauf
0541 JMP $058F    sonst kein SYNC-Signal gefunden, Fehler
0544 LDA $1C01    Byte-Ready
0547 CLV          Leitung freigeben
0548 LDX #$00     Zähler für zu lesende Bytes auf Null
054A BVC $054A    auf Byte warten
054C CLV          Byte-Ready-Leitung löschen
054D LDA $1C01    Byte holen
0550 CMP $05A0,X  mit Wert aus der Tabelle vergleichen
0553 BNE $0587    wenn ungleich, dann Fehler, Rücksprung
0555 INX          sonst Zähler erhöhen
0556 CPX #$06     vergleiche, ob alle Bytes gelesen
0558 BNE $054A    verzweige, wenn noch Bytes zu lesen
055A JSR $05C3    R/W-Kopf um eine Halbspur verschieben
055D LDA #$3F     Zähler für Fehler so stellen, daß kein
055F STA $3B      Fehler mehr erlaubt ist.
0561 JSR $0593    Zähler für Länge des nicht SYNC-Bereichs
0564 NOP          auf 0 stellen
0565 NOP
0566 NOP
0567 LDA $0551    LOW-Byte des Zeigers auf die zu
056A CLC          vergleichenden Daten um
056B ADC #$08     acht erhöhen
056D STA $0551    Wert speichern
0570 CMP #$B8     vergleiche, ob schon alle Bytes verglichen
0572 BNE $0532    verzweige, wenn nicht alles verglichen
0574 JSR $05C0
0577 JSR $05C0    R/W-Kopf auf Ausgangsposition stellen
057A JSR $05C0
057D LDA #$FF     positive Rückmeldung an den Computer
057F STA $0010    Übergeben
0582 STA $0011
0585 BNE $0541    unbedingter Sprung, Rücksprung
0587 INC $3B      Zähler für Fehlversuche erhöhen
0589 LDA $3B      Zähler laden
```

```
058B CMP #$40     mit 64 vergleichen
058D BNE $0532    wenn kleiner, dann erneuter Versuch
058F LDA #$00     sonst negative Rückmeldung an Computer
0591 BEQ $057F    unbedingter Sprung

0593 LDA #$00     Löschen der Zähler
0595 STA $05FE    für die Länge des nicht SYNC-Bereichs
0598 STA $05FF
059B RTS          Rücksprung
059C BRK
059D BRK
059E BRK
059F BRK

05A0 69 A5 5A 96 56 59 A6 A9
05A8 59 9A 6A 65 66 55 99 AA Daten, die mit den auf Disk
05B0 66 95 96 69 A5 5A 6A 6A verglichen werden
05B8 00 00 00 00 00 00 00 00

05C0 LDA #$CA     Wert für Befehl 'DEX' laden
05C2 Byte $2C     Skip nach $05A5
05C3 LDA #$E8     Wert für Befehl 'INX' laden
05C5 STA $05CB    und Wert nach $05AB schreiben
05C8 LDX $1C00
05CB DEX
05CC TXA
05CD AND #$03
05CF STA $4B      R/W-Kopf um eine Halbspur
05D1 LDA $1C00    nach innen oder außen fahren
05D4 AND #$FC
05D6 ORA $4B
05D8 STA $1C00
05DB LDX #$10
05DD LDY #$00
05DF DEY          Zeitschleife, um dem R/W-Kopf
05E0 BNE $05DF    Zeit zu geben, um sich zu positionieren
05E2 DEX
05E3 BNE $05DF
05E5 RTS          Rücksprung
```

Auch hier wieder der entsprechende BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 24740 THEN PRINT"ERROR IN DATAS":STOP
135 PRINT#1,"M-E"CHR$(3)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
170 IFA =255 THENPRINT"SCHUTZ OK":END
180 PRINT"SCHUTZ ERROR":END
302 DATA 76, 25,  5, 32, 66,208,169, 38,133, 10,169,  0,133, 11,169,224,
133,  2
304 DATA 36,  2, 48,252, 76, 66,208,173,  0, 28, 41,159,  9,  8,141,  0,
 28,234
306 DATA234, 32,147,  5,234,169,  0,133, 59, 44,  0, 28, 16,251, 44,  0,
 28, 16
308 DATA 13,238,254,  5,208,246,238,255,  5,208,241, 76,158,253,173,  1,
 28,184
310 DATA162,  0, 80,254,184,173,  1, 28,221,160,  5,208, 50,232,224,  6,
208,240
312 DATA 32,195,  5,169, 63,133, 59, 32,147,  5,234,234,234,173, 81,  5,
 24,105
314 DATA  8,141, 81,  5,201,184,208,190, 32,192,  5, 32,192,  5, 32,192,
  5,169
316 DATA255,141, 16,  0,141, 17,  0,208,186,230, 59,165, 59,201, 64,208,
163,169
318 DATA  0,240,236,169,  0,141,254,  5,141,255,  5, 96,  0,  0,  0,  0,
105,165
320 DATA 90,150, 86, 89,166,169, 89,154,106,101,102, 85,153,170,102,149,
150,105
322 DATA165, 90,106,106,  0,  0,  0,  0,  0,  0,  0,  0,169,202, 44,169,
232,141
324 DATA203,  5,174,  0, 28,202,138, 41,  3,133, 75,173,  0, 28, 41,252,
  5, 75
326 DATA141,  0, 28,162, 10,160,  0,136,208,253,202,208,250, 96, -1
```

Wie bei den anderen von uns vorgestellten Kopierschutzverfahren wird auch hier die entsprechende Rückmeldung der Floppy in Speicherstelle $10 der Floppy abgelegt. Sollte die Speicherstelle den Wert 00 haben, so liegt eine Kopie vor.

6.4 Mit Nullen beschriebene Spuren

Der in diesem Kapitel beschriebene Kopierschutz ist einer von jenen, die bis heute noch nicht kopiert werden können. Die einzige Chance für das Kopierprogramm, ein Duplikat auzufertigen, ist die, daß es den Schutz erkennt, und reproduziert. Kopieren, also die Daten lesen und genauso wieder schreiben, kann man diesen Schutz aufgrund der Hardware der Floppy nicht. Daß das Kopierprogramm den Schutz richtig erkennt, ist ebenfalls so gut wie unmöglich, weshalb sich dieser Kopierschutz, wie auch die anderen in diesem Buch vorgestellten, ausgezeichnet für den professionellen Einsatz nutzen läßt.

Um das System des Schutzes zu verstehen, müssen wir uns näher mit der Lese- und Schreibtechnik des Drive-Controllers beschäftigen.

Wie werden die Daten auf der Magnetschicht der Diskette abgelegt? Die Daten, die beim Schreiben in die Speicherstelle $1C01 geschrieben werden, werden bitweise zum R/W-Kopf geholt, wo sie in Magnetsignale umgewandelt werden. Diese Signale sehen nicht so aus, wie man vielleicht vermutet, nämlich daß ein Null-Bit durch ein nach Norden und ein Eins-Bit durch ein nach Süden gerichtetes Magnetfeld dargestellt wird. Statt dessen wird ein Eins-Bit durch eine Änderung des Magnetfelds und ein Null-Bit durch ein gleichbleibendes Magnetfeld dargestellt. Sie werden sich vielleicht fragen, wie man nach dieser Methode die Null-Bits beim Lesen überhaupt erkennen soll, da doch beim Anlegen eines Null-Bits "nichts passiert". Um diese Bits zu erkennen, läuft wärend des Lesens und Schreibens ein Timer mit, nach dessen Unterlauf ein Bit auf Disk geschrieben oder von Disk gelesen wird. Sollte bis zum Ablauf des Timers ein Magnetwechsel auf der Diskette festgestellt worden sein, wird

ein Eins-Bit vermerkt und der Timer erneut gestartet. Wenn bis zum Ablauf des Timers kein Magnetwechsel vorliegt, wird ein Null-Bit registriert. Aufgrund der Gleichlaufschwankungen des Laufwerks würden schnell Lesefehler auftreten, wenn zu viele Null-Bits hintereinander auf der Diskette stehen, da der Controller sich nur auf seinen Timer verlassen muß und keinen Anhaltspunkt auf der Diskette hat. Aus diesem Grund liegen bei der GCR-Codierung nie mehr als zwei Null-Bits hintereinander.

Den Umstand, daß der Controller beim Finden zu vieler Null-Bits durcheinander kommt, machen wir uns in diesem Kopierschutz zunutze. Wir schreiben mitten in einen Block eine Anzahl von Nullen, was bedeutet, daß sich die Magnetschicht auf dem von uns beschriebenen Stück nicht ändert. Der Controller kann diese Daten jedoch nicht richtig verarbeiten und versucht, sie irgendwie zu entziffern. Bei dieser Entzifferung fügt er jedoch Eins-Bits ein, die er eigentlich gar nicht gelesen hat. Wenn er versucht, dieses Stück erneut zu lesen, weichen die vermeintlich erkannten Daten jedoch stark von den zuvor gelesenen ab. Das Kopierprogramm, das diesen Block liest, erhält ebenfalls irgendwelche "Fantasiewerte", welche es beim Schreiben auch so auf Diskette speichert. Wenn man jetzt diese vom Kopierprogramm geschriebenen Daten mehrmals einliest, weichen diese nicht voneinander ab, da für den Drive-Controller richtige Daten geschrieben wurden und somit auch fehlerfrei gelesen werden können. Sie werden uns beipflichten, daß es sich bei diesem Kopierschutz um ein sehr sicheres System handelt.

Nachdem wir nun die theoretischen Voraussetzungen geschaffen haben, wollen wir in die Praxis Übergehen. Das folgende Maschinensprachprogramm erzeugt einen zuvor beschriebenen Kopierschutz auf dem eingestellten Sektor. Das Programm wird ab $0552 gestartet.

```
0500 JSR $F510    Blockheader suchen
0503 LDX #$09     Zähler für zu überlesende Byte-Anzahl
0505 BVC $0505    warten, bis ein Byte anliegt
0507 CLV          Byte-Ready-Leitung löschen
0508 DEX          Zähler verringern
```

```
0509 BNE $0505   verzweige, wenn Zähler nicht abgelaufen
050B LDA #$FF    Port auf Ausgang stellen
050D STA $1C03   um R/W-Kopf auf Schreiben umzustellen
0510 LDA $1C0C   R/W-Kopf auf
0513 AND #$1F    Schreiben stellen
0515 ORA #$C0
0517 STA $1C0C
051A LDA #$FF    $FF zum Schreiben der SYNC-Markierung
051C LDX #$05    Zähler für Länge der SYNC-Markierung
051E STA $1C01   SYNC-Markierung schreiben
0521 CLV         Byte-Ready-Leitung löschen
0522 BVC $0522   warten, bis Byte geschrieben
0524 CLV         Byte-Ready-Leitung löschen
0525 DEX         Zähler für SYNC verringern
0526 BNE $0522   verzweige, wenn nicht abgelaufen
0528 LDA #$55    $55 laden
052A STA $1C01   und auf Disk schreiben
052D BVC $052D   warten, bis Byte geschrieben
052F CLV         Byte-Ready-Leitung löschen
0530 LDA #$CE    $CE laden
0532 STA $1C01   und auf Disk schreiben
0535 BVC $0535   warten, bis Byte geschrieben
0537 CLV         Byte-Ready-Leitung löschen
0538 INX         Zähler für zu schreibende Bytes erhöhen
0539 CPX #$20    vergleiche, ob alle Bytes geschrieben
053B BNE $0535   verzweige, wenn noch Bytes zu schreiben
053D LDA #$00    Akku mit $00 laden
053F LDX #$20    Zählwert für Byte-Anzahl auf 32
0541 STA $1C01   $00 auf Disk schreiben
0544 BVC $0544   warten bis Byte geschrieben
0546 CLV         Byte-Ready-Leitung löschen
0547 DEX         Zähler verringern
0548 BNE $0544   verzweige, wenn noch Bytes zu schreiben
054A JSR $FE00   R/W-Kopf auf Lesen stellen
054D LDA #$01    OK Rückmeldung laden
054F JMP $F969   und Job beenden, Rückmeldung ausgeben
```

Hier wird das Programm gestartet:

```
0552 LDA #$01     Track-Nummer laden
0554 STA $0A      und an Job übergeben
0556 LDA #$00     Sektor-Nummer laden
0558 STA $0B      und an Job übergeben
055A LDA #$E0     Job-Code 'E0' (Programm ausführen)
055C STA $02      in Job-Speicher schreiben
055E LDA $02      Rückmeldung abwarten
0560 BMI $055E    verzweige, wenn noch keine Rückmeldung
0562 RTS          Rücksprung
```

Anschließend der entsprechende BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 12308 THEN PRINT"ERROR IN DATAS":STOP
105 INPUT "WELCHER TRACK";T
110 INPUT "WELCHER SEKTOR";S
120 PRINT#1,"M-W"CHR$(83)CHR$(5)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(87)CHR$(5)CHR$(1)CHR$(S)
135 PRINT#1,"M-E"CHR$(82)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(2)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
160 IF A >127 THEN 145
170 IFA =1 THENPRINT"OK":END
180 PRINT"ERROR":END
302 DATA 32, 16,245,162,  9, 80,254,184,202,208,250,169,255,141,  3, 28,
173, 12
304 DATA 28, 41, 31,  9,192,141, 12, 28,169,255,162,  5,141,  1, 28,184,
 80,254
306 DATA184,202,208,250,169, 85,141,  1, 28, 80,254,184,169,206,141,  1,
 28, 80
308 DATA254,184,232,224, 32,208,248,169,  0,162, 32,141,  1, 28, 80,254,
184,202
```

```
310 DATA208,250, 32,  0,254,169,  1, 76,105,249,169, 18,133, 10,169,  0,
133, 11
312 DATA169,224,133,  2,165,  2, 48,252, 96, -1
```

Die Daten, die vor den Nullen auf Disk geschrieben werden, sind so gewählt, daß kein '22er Read-Error' entsteht. Ins Bin-Format umgesetzt steht an der ersten Stelle nach der SYNC-Markierung eine $07 (Kennzeichen für Datenblock).

Die Abfrage des Kopierschutzes ist noch einfacher als das Auftragen. Die Daten des geschützten Blocks werden zweimal gelesen, in der Floppy gespeichert und daraufhin miteinander verglichen. Sollten die Daten nicht übereinstimmen, liegt keine Kopie vor. Die Übereinstimmung der Daten ist das Kennzeichen der Kopie. Die Rückmeldung an den Computer wird in $10 gespeichert. $FF bedeutet positive und $00 negative Rückmeldung.

Das Floppy-Programm liegt wieder ab $0500 und wird ab $0543 gestartet.

```
0500 LDA #$05     Zähler für Fehlversuche
0502 STA $37      setzen
0504 JSR $F50A    entsprechenden Datenblock suchen
0507 LDX #$00     Zähler für die zu lesenden Bytes auf Null
0509 BVC $0509    warten, bis Byte gelesen
050B CLV          Byte-Ready-Leitung löschen
050C LDA $1C01    Byte vom Port holen
050F STA $0300,X  und speichern
0512 INX          Zähler erhöhen
0513 BNE $0509    verzweige, wenn nicht alle Bytes geholt
0515 JSR $F50A    gleichen Datenblock erneut holen
0518 LDX #$00     Zähler auf 00 setzen
051A BVC $051A    auf Byte warten
051C CLV          Byte-Ready-Leitung löschen
051D LDA $1C01    Byte vom Port holen
0520 STA $0400,X  und speichern
0523 INX          Zähler erhöhen
0524 BNE $051A    verzweige, wenn nicht alle Bytes geholt
0526 LDA $0300,X  Bytes des zuerst geladenen Blocks mit
```

```
0529 CMP $0400,X denen des zweiten vergleichen
052C BNE $053C   verzweige, wenn ungleich
052E INX         sonst Zähler erhöhen
052F BNE $0526   verzweige, wenn nicht alle Bytes
                 verglichen
0531 DEC $37     Fehlerzähler verringern
0533 BNE $0504   verzweige, wenn erneuter Versuch erlaubt
0535 LDA #$00    sonst negative Rückmeldung
0537 STA $10     übergeben
0539 JMP $FD9E   Rücksprung
053C LDA #$FF    positive Rückmeldung laden
053E STA $10     und übergeben
0540 JMP $FD9E   Rücksprung
```

Das Programm wird hier gestartet:

```
0543 LDA #$01    Track-Nummer laden
0545 STA $0A     und übergeben
0547 LDA #$00    Sektor-Nummer laden
0549 STA $0B     und übergeben
054B LDA #$E0    Job-Code 'E0' laden
054D STA $02     und an den Job-Speicher übergeben
054F LDA $02     auf Rückmeldung warten
0551 BMI $054F   verzweige, wenn noch keine Rückmeldung
0553 RTS         Rücksprung
```

Folgend wieder unser kleiner BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 9963 THEN PRINT"ERROR IN DATAS":STOP
105 INPUT "WELCHER TRACK";T
110 INPUT "WELCHER SEKTOR";S
120 PRINT#1,"M-W"CHR$(68)CHR$(5)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(72)CHR$(5)CHR$(1)CHR$(S)
```

```
135 PRINT#1,"M-E"CHR$(67)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
170 IFA =255 THENPRINT"SCHUTZ OK":END
180 PRINT"SCHUTZ ERROR":END
302 DATA169,  5,133, 55, 32, 10,245,162,  0, 80,254,184,173,  1, 28,157,
  0,  3
304 DATA232,208,244, 32, 10,245,162,  0, 80,254,184,173,  1, 28,157,  0,
  4,232
306 DATA208,244,189,  0,  3,221,  0,  4,208, 14,232,208,245,198, 55,208,
207,169
308 DATA  0,133, 16, 76,158,253,169,255,133, 16, 76,158,253,169,  1,133,
 10,169
310 DATA  0,133, 11,169,224,133,  2,165,  2, 48,252, 96, -1
```

6.5 Überprüfen eines kompletten Tracks

Das Überprüfen jedes Bytes eines gesamten Tracks dient auch zur Erstellung eines sehr guten Kopierschutzes, der bisher nicht kopiert werden kann. Dieses Schutzsystem wird auch jetzt noch von großen Firmen mit Erfolg eingesetzt.

Warum Kopierprogramme es nicht schaffen, diesen Track zu kopieren, liegt daran, daß die Tracks beim Schreiben nicht gleich lang sind und darum nicht exakt dieselbe Anzahl von Bytes enthalten. Die variable Byte-Anzahl eines Tracks kommt durch die Schwankungen des Laufwerkmotors zustande.

Die nächste Hürde, die Kopierprogramme zu überwinden haben, ist die Tatsache, daß das letzte Daten-Byte sich nahtlos an die SYNC-Markierung anschließt. Selbst wenn das Kopierprogramm alle Daten richtig erkennt und versucht, das Ende der Daten korrekt an den Anfang der Daten zu bringen, ist es sehr leicht möglich, daß beim Umschalten des R/W-Kopfes auf den Lesebetrieb in die SYNC-Markierung noch Null-Bits geschrieben werden, welche das Abfrageprogramm erkennt und signalisiert, daß eine Kopie vorliegt.

Mit dem Problem, daß beim Umschalten des Kopfes vom Schreib- in den Lesebetrieb die SYNC-Markierung "beschädigt" werden kann, müssen wir in unserem Schutzauftragungsprogramm auch fertig werden. Wir lösen das Problem durch mehrere Schreibversuche. Es wird so lange geschrieben, bis der Kopierschutz fehlerfrei aufgetragen wurde.

Im folgenden gehen wir näher auf unser Lese- und Schreibprogramm ein.

Zu Beginn des Auftragens des Schutzes wird der angegebene Track mit SYNCs gelöscht. Daraufhin werden die Daten auf den Track geschrieben. In unserem Fall handelt es sich um die immer im Wechsel geschriebenen Daten $45 und $79. Nachdem wir diese Zahlen ca. 6000mal geschrieben haben, folgt eine $35 als Endmarkierung, eine $66 und danach wieder die SYNC-Markierung.

Das Abfrageprogramm wartet auf die SYNC-Markierung und liest die Daten, die auch auf ihren Wert geprüft werden, bis es auf die $35 stößt und die Anzahl der bisher geschriebenen Daten kontrolliert. Nun werden weitere sechs Bytes geholt. Das erste Byte muß $66 sein. Sollte die SYNC-Markierung korrekt sein, so wird noch ein $FF (Anfang der SYNC-Markierung) gefunden und danach das Lesen von Daten hardwaremäßig unterbunden, da eine SYNC-Markierung anliegt. Folglich sind die als nächstes gelesenen Daten wieder die Anfangs-Bytes.

Bei einer nicht korrekten SYNC-Markierung, die durch das Umschalten vom Schreib- auf den Lesebetrieb zerstört wurde, wird beim Lesen vor dem Auffinden der Anfangsdaten noch ein Byte gefunden, was wieder auf eine Kopie schließen läßt.

Nachdem wir jetzt näher auf die Programme eingegangen sind, stellen wir sie Ihnen jetzt in Form der Listings vor. Als erstes folgt das Kopierschutzerstellungs-Programm, das ab $0571 gestartet wird.

```
0500 LDA $1C00   Control-Port laden
0503 AND #$9F    Speed auf 00 stellen
0505 ORA #$08    und LED anschalten
0507 STA $1C00   und Wert speichern
050A JSR $FDA3   Track mit SYNC löschen
050D LDX #$02    Zähler für Byte
050F LDY #$18    Anzahl setzen
0511 LDA #$45    erstes zu schreibendes Byte
0513 STA $1C01   zum R/W-Kopf schicken
0516 BVC $0516   warten, bis Byte geschrieben
0518 CLV         Byte-Ready-Leitung löschen
0519 INX         Zähler erhöhen
051A LDA #$79    zweites zu schreibendes Byte laden
051C STA $1C01   und zu R/W-Kopf schicken
051F BVC $051F   warten, bis Byte geschrieben
0521 CLV         Byte-Ready-Leitung löschen
0522 INX         Zähler erhöhen
0523 BNE $0511   weiter, wenn kein Überlauf
0525 DEY         sonst HIGH-Byte des Zählers verringern
0526 BNE $0511   verzweige, wenn noch Bytes zu schreiben
0528 LDA #$35    Markierungs-Byte laden
052A STA $1C01   und auf Disk schreiben
052D BVC $052D   warten, bis Byte geschrieben
052F CLV         Byte-Ready-Leitung löschen
0530 LDA #$66    $66 laden
0532 STA $1C01   und auf Disk schreiben
0535 BVC $0535   warten, bis Byte geschrieben
0537 CLV         Byte-Ready-Leitung löschen
0538 LDA #$FF    $FF für SYNC-Markierung laden
053A STA $1C01   und schreiben
053D BVC $053D   warten, bis Byte geschrieben
053F CLV         Byte-Ready-Leitung löschen
0540 JSR $FE00   Kopf auf Lesen umstellen
```

Der folgende Programmteil kontrolliert, ob in der SYNC-Markierung beim Umschalten des Kopfes auf den Lesebetrieb kein Fehler entstanden ist.

```
0543 LDA $1C00   Control-Port laden
0546 BMI $0543   und auf SYNC-Markierung warten
0548 LDA $1C01   Port wieder freimachen
054B CLV         Byte-Ready-Leitung löschen
054C BVC $054C   Warten, bis Byte gelesen
054E CLV         Byte-Ready-Leitung löschen
054F LDA $1C01   Byte vom Port holen
0552 CMP #$35    mit Endmarkierung vergleichen
0554 BNE $054C   warten, bis Byte gefunden
0556 LDX #$00    Zähler für zu lesende Bytes laden
0558 BVC $0558   warte, bis Byte anliegt
055A CLV         Byte-Ready-Leitung löschen
055B LDA $1C01   Byte vom Port holen
055E CMP $056B,X mit Bytes aus Tabelle vergleichen
0561 BNE $050A   erneut schreiben, wenn ungleich
0563 INX         Zähler erhöhen
0564 CPX #$06    vergleichen, ob alle Bytes verglichen
0566 BNE $0558   verzweige, wenn noch Bytes zu holen
0568 JMP $FD9E   Rücksprung

056B 66 FF 45 79 45 79  Bytes, mit denen verglichen wird

0571 LDA #$01    Track-Nummer laden
0573 STA $0A     und übergeben
0575 LDA #$E0    Job-Code 'E0' fahren
0577 STA $02     und übergeben
0579 LDA $02     auf Rückmeldung warten
057B BMI $0579   warten, bis Rückmeldung erhalten
057D RTS         Rücksprung
```

Nachfolgend der zugehörige BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 15481 THEN PRINT"ERROR IN DATAS":STOP
110 INPUT "TRACK-NUMMER";T
```

```
120 PRINT#1,"M-W"CHR$(114)CHR$(5)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(113)CHR$(5)
140 PRINT#1,"I"
150 CLOSE1
302 DATA173,  0, 28, 41,159,  9,  8,141,  0, 28, 32,163,253,162,  2,160,
 24,169
304 DATA 69,141,  1, 28, 80,254,184,232,169,121,141,  1, 28, 80,254,184,
232,208
306 DATA236,136,208,233,169, 53,141,  1, 28, 80,254,184,169,102,141,  1,
 28, 80
308 DATA254,184,169,255,141,  1, 28, 80,254,184, 32,  0,254,173,  0, 28,
 48,251
310 DATA173,  1, 28,184, 80,254,184,173,  1, 28,201, 53,208,246,162,  0,
 80,254
312 DATA184,173,  1, 28,221,107,  5,208,167,232,224,  6,208,240, 76,158,
253,102
314 DATA255, 69,121, 69,121,169,  1,133, 10,169,224,133,  2,165,  2, 48,
252, 96
316 DATA -1
```

Wundern Sie sich nicht, wenn das Auftragen des Schutzes etwas länger dauert, denn das Auftragungsprogramm schreibt so lange, bis der Schutz korrekt aufgetragen ist.

Als nächstes folgt das Assemblerlisting des Abfrageprogramms, das ab $0561 gestartet wird:

```
0600 LDA $1C00   Control-Port laden
0603 AND #$9F    Speed auf Null stellen
0605 ORA #$08    und LED
0607 STA $1C00   einschalten
060A LDX #$00    Zähler für Anzahl
060C LDY #$00    der Bytes auf Null setzen
060E LDA $1C00   Control-Port laden
0611 BMI $060E   und warten, bis SYNC-Markierung gefunden
0613 LDA $1C01   Port freimachen
0616 CLV         Byte-Ready-Leitung löschen
0617 BVC $0617   warten, bis Byte gelesen
0619 CLV         Byte-Ready-Leitung löschen
```

```
061A LDA $1C01   Byte vom Port holen
061D CMP #$45    und mit $45 vergleichen
061F BNE $0632   Fehler, wenn ungleich
0621 INX         Byte-Zähler erhöhen
0622 BVC $0622   warten, bis Byte anliegt
0624 CLV         Byte-Ready-Leitung löschen
0625 LDA $1C01   Byte vom Port holen
0628 CMP #$79    mit $79 vergleichen
062A BNE $0632   Fehler, wenn ungleich
062C INX         Byte-Zähler erhöhen
062D BNE $0617   verzweige, wenn kein Überlauf
062F INY         HIGH-Byte des Zählers erhöhen
0630 BNE $0617   verzweige, wenn kein Überlauf
0632 CMP #$35    Byte mit Markierungs-Byte vergleichen
0634 BNE $0657   Fehler, wenn ungleich
0636 CPX #$FE    LOW-Byte des Zählers mit $FE vergleichen
0638 BNE $0657   Fehler, wenn ungleich
063A CPY #$17    HIGH-Byte des Zählers mit $17 vergleichen
063C BNE $0657   Fehler, wenn ungleich
063E LDX #$00    Zähler für zu lesende Bytes löschen
0640 BVC $0640   warten, bis Byte gelesen
0642 CLV         Byte-Ready-Leitung löschen
0643 LDA $1C01   Byte vom Port holen
0646 CMP $065B,X mit Byte aus Tabelle vergleichen
0649 BNE $0657   Fehler, wenn ungleich
064B INX         Zähler für Byte-Anzahl erhöhen
064C CPX #$06    vergleiche, ob alle Bytes geholt
064E BNE $0640   weiter, wenn noch Bytes zu lesen
0650 LDA #$FF    positive Rückmeldung laden
0652 STA $10     und übergeben
0654 JMP $FD9E   Rücksprung
0657 LDA #$00    negative Rückmeldung laden
0659 BEQ $0652   unbedingter Sprung

065B 66 FF 45 79 45 79   Bytes, mit denen verglichen wird

0661 LDA #$01    Track-Nummer laden
0663 STA $0C     und an Job übergeben
0665 LDA #$E0    Job-Code 'E0' laden
0667 STA $03     und in Job-Speicher schreiben
```

```
0669 LDA $03      überprüfen, ob Rückmeldung erhalten
066B BMI $0669    verzweige, wenn noch keine Rückmeldung
066D RTS          Rücksprung
```

Im Anschluß an das Assemblerlisting folgt wie üblich der BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 13656 THEN PRINT"ERROR IN DATAS":STOP
110 INPUT "TRACK-NUMMER";T
120 PRINT#1,"M-W"CHR$(98)CHR$(6)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(97)CHR$(6)
140 FORN=1 TO 500:NEXT
150 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
160 GET#1,A$
170 IF ASC(A$+CHR$(0))<>255 THENPRINT" SCHUTZ ERROR":END
180 PRINT" SCHUTZ OK"
302 DATA173,  0, 28, 41,159,  9,  8,141,  0, 28,162,  0,160,  0,173,  0,
 28, 48
304 DATA251,173,  1, 28,184, 80,254,184,173,  1, 28,201, 69,208, 17,232,
 80,254
306 DATA184,173,  1, 28,201,121,208,  6,232,208,232,200,208,229,201, 53,
208, 33
308 DATA224,254,208, 29,192, 23,208, 25,162,  0, 80,254,184,173,  1, 28,
221, 91
310 DATA  6,208, 12,232,224,  6,208,240,169,255,133, 16, 76,158,253,169,
  0,240
312 DATA247,102,255, 69,121, 69,121,169,  1,133, 12,169,224,133,  3,165,
  3, 48
314 DATA252, 96, -1
```

6.6 SYNC-Manipulationen

Eine recht gute Möglichkeit zur Erstellung eines Kopierschutzes, ergibt sich aus der Änderung der SYNC-Makierungen auf einem Track. Wie man mit Hilfe von veränderten SYNC-Markierungen einen Kopierschutz aufbauen kann, wollen wir in den folgenden Kapiteln zeigen.

6.6.1 Killertracks

Killertrack! Dieser Name hört sich ziemlich bedrohlich an, aber was steckt eigentlich dahinter? Killertracks sind nichts anderes als mit SYNC-Markierungen beschriebene Tracks.

Ein ausschließlich mit SYNC-Markierungen beschriebener Track brachte die früheren Kopierprogramme oder jedes andere Programm, mit dem man versucht, von einen solchen Track Daten zu lesen, unweigerlich zum Absturz, woraus sich wohl auch der Name ableiten läßt. Die Erklärung für dieses Verhalten ist leicht zu geben, doch um sie zu verstehen, müssen wir noch einmal daran erinnern, wie die normale Routine zum Lesen eines Bytes von Diskette aussieht.

```
    CLV         Byte-Ready-Leitung löschen
L1  BVC L1      warten, bis Byte anliegt
    LDA $1C01   Byte vom Port holen
```

Meistens wird vor dem Lesen eines Bytes noch auf die SYNC-Markierung gewartet, die bei einem Killertrack natürlich sofort gefunden wird. Nachdem der Controller der Floppy eine SYNC-Markierung erkannt hat, wird das Lesen von Daten automatisch hardwaremäßig unterbrochen und erst nach Ende des Signals wieder freigegeben. Die Byte-Ready-Leitung ist somit während des Anliegens einer SYNC-Markierung gesperrt. Wenn wir uns jetzt das kleine Teilprogramm ansehen, so erkennen wir, daß das Programm zwangsläufig in einer Endlosschleife laufen muß.

Die Kopierprogramme, die die Diskette nicht auf Existenz eines solchen Tracks überprüfen, werden automatisch "gekillt".

Kopierprogramme heutigen Standards lassen sich von reinen Killertracks nicht mehr beeindrucken, verarbeiten aber Abwandlungen eines solchen Tracks nicht so souverän, wie man es erwarten könnte.

Zum Auftragen eines Killertracks reicht ein Aufruf einer Unterroutine des Formatprogramms, welche einen Track wie beschrieben mit SYNCs löscht. Diese Routine läßt sich mit 'JMP $FDA3' in der Floppy aufrufen.

6.6.2 Verlängerte SYNC-Markierungen

Einen sehr einfachen, aber trotzdem sehr guten Kopierschutz erhält man durch das Einsetzen von leicht veränderten Killertracks. Diese Veränderung sieht so aus, daß man einen Track mit SYNCs löscht, um danach einige wenige Daten auf den Track zu schreiben. Das Abfrageprogramm prüft sowohl die Richtigkeit der Daten als auch die Länge der SYNC-Markierung. Uns ist kein Programm bekannt, das es schafft, einen solchen Track zu kopieren, was weniger an der Raffiniertheit des Schutzes als an der mangelnden Qualität der Kopierprogramme liegt. Die Kopierprogramme, die es schaffen, die Daten, die auf den Track geschrieben wurden, zu kopieren, versagen, wenn es um die Länge der SYNC-Markierung geht, und diejenigen, die es in etwa schaffen, die Länge der SYNC-Markierung zu kopieren, kopieren die Daten nicht richtig. Das Ergebnis ist, daß der Schutz trotz seiner Einfachheit nicht kopiert werden kann.

Es folgt das Programm, mit dem Sie den Schutz auftragen können. Es wird ab $0527 gestartet:

```
0500 LDA $1C00   Control-Port laden
0503 AND #$9F    Speed auf 0 stellen
0505 ORA #$08    und LED anschalten
0507 STA $1C00   Wert speichern
```

```
050A JSR $FDA3    Track mit SYNC löschen
050D LDX #$00     Zähler für Byte-Anzahl auf Null stellen
050F LDA $0523,X  Bytes holen
0512 BVC $0512    warten, bis Byte geschrieben wird
0514 CLV          Byte-Ready-Leitung löschen
0515 STA $1C01    Byte zum R/W-Kopf schicken
0518 INX          Zähler erhöhen
0519 CPX #$06     vergleiche, ob alle Bytes geschrieben
051B BNE $050F    verzweige, wenn noch Bytes zu schreiben
051D JSR $FE00    R/W-Kopf auf Lesen schalten
0520 JMP $FD9E    Rücksprung

0523 53 54 55 56 A9  Die zu schiebenden Bytes

0527 LDA #$24     Track-Nummer laden
0529 STA $0A      und übergeben
052B LDA #$E0     Job-Code 'E0' laden
052D STA $02      und in Job-Speicher schreiben
052F LDA $02      Rückmeldung abwarten
0531 BMI $052F    verzweige, wenn keine Rückmeldung
0533 RTS          Rücksprung
```

Im Anschluß folgt wieder unserer BASIC-Loader:

```
10 OPEN1,8,15
20 READ X:IF X=-1THEN 100
30 SU=SU+X:PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IF SU <> 5576 THENPRINT"FEHLER IN DATAS":STOP
110 INPUT "WELCHER TRACK";T
120 PRINT#1,"M-W"CHR$(40)CHR$(5)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(39)CHR$(5)
140 CLOSE1
302 DATA173,  0, 28, 41,159,  9,  8,141,  0, 28, 32,163,253,162,  0,189,
 35,  5
304 DATA 80,254,184,141,  1, 28,232,224,  6,208,242, 32,  0,254, 76,158,
253, 83
306 DATA 84, 85, 86,169, 36,133, 10,169,224,133,  2,165,  2, 48,252, 96,
 -1
```

Folgend finden Sie das entsprechende Programm zur Schutzabfrage. Als erstes stellen wir Ihnen das Maschinen-Listing vor. Es liegt ab $0600 im RAM der Floppy und wird bei $064A gestartet.

```
0600 LDA $1C00    Control-Port laden
0603 AND #$9F     Speed auf 0 stellen
0605 ORA #$08     und LED anschalten
0607 STA $1C00    Wert wieder speichern
060A LDA $1C00    Control-Port laden
060D BMI $060A    warte, bis SYNC gefunden
060F CLV          Byte-Ready-Leitung löschen
0610 LDA $1C01    Byte vom Port holen
0613 LDX #$00     Zähler für Bytes auf 0 stellen
0615 BVC $0615    warte, bis Byte anliegt
0617 CLV          Byte-Ready-Leitung löschen
0618 LDA $1C01    Byte vom Port holen
061B CMP $0646,X  Byte mit Tabelle vergleichen
061E BNE $0642    verzweige, wenn Bytes ungleich (Fehler)
0620 INX          Zähler erhöhen
0621 CPX #$05     vergleiche, ob alle Bytes gelesen
0623 BNE $0615    verzweige, wenn noch Bytes zu lesen
0625 LDA $1C00    Control-Port laden
0628 BMI $0625    warte, bis SYNC gefunden
062A LDX #$00     Zähler für
062C LDY #$00     SYNC Länge löschen
062E INX          Zähler erhöhen
062F BNE $0632    verzweige, wenn kein Überlauf
0631 INY          HIGH-Byte erhöhen
0632 LDA $1C00    Control-Port laden
0635 BPL $062E    verzweige, wenn SYNC noch anliegt
0637 CPY #$38     HIGH-Byte mit 38 vergleichen
0639 BCC $0642    Fehler, wenn Wert kleiner
063B LDA #$FF     positive Rückmeldung laden
063D STA $10      und übergeben
```

```
063F JMP $FD9E   Rücksprung
0642 LDA #$00    negative Rückmeldung laden
0644 BEQ $063D   unbedingter Sprung

0646 53 54 55 56 A9  Bytes mit denen verglichen wird

064A LDA #$24    Track-Nummer laden
064C STA $0C     und übergeben
064E LDA #$E0    Job-Code 'E0' laden
0650 STA $03     und in  Job-Speicher schreiben
0652 LDA $03     Job-Speicher laden
0654 BMI $0652   warte, bis Job ausgeführt
0656 RTS         Rücksprung
```

Als nächstes folgt der zugehörige BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 9624 THEN PRINT"ERROR IN DATAS":STOP
110 INPUT" WELCHER TRACK";T
120 PRINT#1,"M-W"CHR$(75)CHR$(6)CHR$(1)CHR$(T)
135 PRINT#1,"M-E"CHR$(74)CHR$(6)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC (A$+CHR$(0))
170 IFA =255 THENPRINT"SCHUTZ OK":END
180 PRINT"SCHUTZ ERROR":END
302 DATA173,  0, 28, 41,159,  9,  8,141,  0, 28,173,  0, 28, 48,251,184,
173,  1
304 DATA 28,162,  0, 80,254,184,173,  1, 28,221, 70,  6,208, 34,232,224,
  5,208
306 DATA240,173,  0, 28, 48,251,162,  0,160,  0,232,208,  1,200,173,  0,
 28, 16
308 DATA247,192, 56,144,  7,169,255,133, 16, 76,158,253,169,  0,240,247,
 83, 84
310 DATA 85, 86,169, 34,133, 12,169,224,133,  3,165,  3, 48,252, 96, -1
```

6.6.3 Der 3000-SYNC-Schutz

Ein weiteres hervorragendes Schutzsystem, das auf der Änderung der SYNC-Markierung basiert, soll hier besprochen werden. Es handelt sich dabei um das Anlegen von zu vielen kurzen SYNC-Markierungen. Aufgrund ihrer Kürze und ihrer Dichte passen bei diesem System über 3000 SYNC-Markierungen auf einen Track (bei einem normalen Track sind es maximal 42). Schon anhand dieser Aussage werden Sie sich vorstellen können, daß ein Kopierprogramm mit diesem Track Probleme bekommen kann. Die SYNC-Markierungen sind so angelegt, daß zwischen den einzelnen Markierungen nur eine 13 Bit lange Nicht-SYNC-Zone liegt. Die Zonen zwischen den SYNC-Markierungen sind bis auf eine gleich. Diese eine abweichende Lücke dient zur Orientierung auf dem Track. Um den Schutz zu prüfen, wird gewartet, bis die erwähnte abweichende Lücke gefunden wird. Daraufhin wird die Anzahl der SYNC-Markierungen gezählt, bis der Ausgangspunkt wieder erreicht ist. Die Anzahl der gefundenen SYNC-Markierungen darf nicht von der beim Auftragen des Schutzes gezählten Anzahl abweichen. Warum es für das Kopierprogramm so gut wie unmöglich ist, den Track zu kopieren, ist recht einleuchtend. Aufgrund der Laufwerkschwankungen (die nicht zu vermeiden sind) hat jeder Track beim erneuten Schreiben desselben eine andere Anzahl von Bytes. Das zweite Problem, das das Kopierprogramm zu lösen hat, ist, einen Anhaltspunkt auf diesem Track zu finden, denn die kleine Abweichung in der Lücke, die uns als Orientierung dient, ist schwer zu finden, vor allem, wenn man nicht weiß, wonach man suchen soll.

Bis heute ist kein Kopierprogramm in der Lage, einen solchen Track originalgetreu zu kopieren.

Nachdem wir uns mit dem Abfragen des Kopierschutzes befaßt haben, wenden wir uns dem Auftragen des Schutzes zu. Beim Erstellen des Tracks ist einiges zu beachten. Aufgrund der Tatsache, daß die Byte-Anzahl beim Schreiben variiert, wird ein Zählen der SYNC-Markierungen auch schon beim Auftragen des

Schutzes nötig, um die entsprechende Anzahl danach an das Prüfprogramm zu übergeben. Zusätzlich zu der Anzahl der SYNCs muß noch unser Markierungs-Byte bei jedem Auftragen des Schutzes neu ermittelt werden. Dieses Byte ist ebenfalls variabel.

Durch die sich bei jedem Auftragen des Schutzes ändernden Parameter sind wir bei diesem System andere Wege gegangen, um ein komfortables Arbeiten zu gewährleisten.

Das folgende BASIC-Programm ermöglicht die Wahl des Tracks, auf dem der Schutz aufgetragen werden soll. Anders als bei den bisherigen Beispielen wird es nicht nötig sein, die Kopierschutzabfrage bei jedem Abfragen in die Floppy zu schicken. Stattdessen wird sie hier auf einem zuvor wählbaren Track und Sektor gespeichert. Um den Kopierschutz abzufragen, muß lediglich der entsprechende Sektor in den Puffer der Floppy geladen werden, um dort mit 'M-E' ab $0503 gestartet zu werden. Die Rückmeldung nach der Überprüfung des Schutzes wird auch, wie bei den anderen Beispielen, in $10 der Floppy abgelegt. Analog zu den bisherigen Programmen bedeutet die Rückmeldung $00, daß eine Kopie vorlag. $FF kennzeichnet das 'Original'.

Doch jetzt wollen wir uns nach dieser ausführlichen Einleitung das BASIC-Programm ansehen, das wie beschrieben den Kopierschutz sowie die Abfrage auf Diskette installiert.

```
10 PRINTCHR$(147);"BITTE WARTEN"
20 FORI=1TO306STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
30 A=ASC(A$)-48:IFA>9THENA=A-7
40 B=ASC(B$)-48:IFB>9THENB=B-7
50 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A:NEXT:READA:IFC= A THENC=0:NEXT
:GOTO70
60 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
70 PRINTCHR$(147)CHR$(17)CHR$(17):INPUT" ERROR TRACK";T
80 IFT>41ORT<1THENPRINTCHR$(145)CHR$(145)CHR$(145):GOTO70
90 PRINT:PRINT:INPUT" TRACK,BLOCK DER KONTROLLE";TK,BK
100 PRINT:PRINT:PRINT" EINGABEN IN ORDNUNG?"
```

```
110 GETA$:IFA$=""THEN110
120 IFA$="N"THENFORK=1TO 8:PRINT CHR$(147):NEXT:GOTO70
130 PRINT:PRINT:PRINT" DISKETTE EINLEGEN"
140 GETA$:IFA$=""THEN140
150 PRINTCHR$(147);"BITTE WARTEN"
160 OPEN1,8,15
170 POKE49159,T:POKE49350,T
180 FORN=0TO191:PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$( PEEK(49152+N)):N
EXT
190 PRINT#1,"M-E"CHR$(3)CHR$(5)
200 PRINT#1,"M-R"CHR$(254)CHR$(5)CHR$(2)
210 GET#1,A$:POKE49437,ASC(A$+CHR$(0))
220 GET#1,A$:POKE49441,ASC(A$+CHR$(0))
230 PRINT#1,"M-R"CHR$(128)CHR$(5)CHR$(1)
240 GET#1,A$:POKE49394,ASC(A$+CHR$(0))
250 POKE49417,ASC(A$+CHR$(0))
260 FORN=0TO116:PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$( PEEK(49343+N)):N
EXT
270 OPEN2,8,2,"#2"
280 PRINT#1,"U2 2 0";STR$(TK);STR$(BK)
290 PRINT#1,"U1 2 0";STR$(TK);STR$(BK)
300 PRINT#1,"M-E"CHR$(3)CHR$(5)
310 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
320 PRINTCHR$(147)
330 GET#1,A$:IFASC(A$+CHR$(0))<>255THENPRINT:PRINT"  SCHUTZ ERROR":GOTO3
50
340 PRINT:PRINT:PRINT:PRINT" SCHUTZ OK"
350 PRINT:PRINT:PRINT" WEITERE DISKETTE?"
360 GETA$:IFA$=""THEN360
370 IFA$="J"THEN GOTO70
380 END
390 PRINT:PRINT:PRINT" ERNEUTER VERSUCH?"
400 GETA$:IFA$=""THEN400
410 IFA$="J"THEN160
420 DATA20,13,05,20,42,D0,A9,24,85,0A,A9,E0,85,02,A5,  123
430 DATA02,30,FC,60,AD,00,1C,29,9F,09,08,8D,00,1C,20,  249
440 DATA0E,FE,A0,10,A2,00,A9,27,8D,01,1C,50,FE,B8,A9,  135
450 DATAFF,8D,01,1C,50,FE,B8,CA,D0,ED,88,D0,EA,A9,FF,  32
460 DATA8D,01,1C,50,FE,B8,50,FE,B8,50,FE,B8,A9,33,8D,  37
470 DATA01,1C,50,FE,B8,20,00,FE,AD,00,1C,30,FB,B8,AD,  154
```

```
480 DATA01,1C,A2,00,50,FE,B8,AD,01,1C,C9,27,F0,EB,8D, 231
490 DATA80,05,8D,97,05,AD,00,1C,30,FB,B8,AD,01,1C,A2, 198
500 DATA00,50,FE,B8,AD,01,1C,C9,37,D0,EB,A2,00,A0,00, 205
510 DATAAD,00,1C,30,FB,B8,AD,01,1C,50,FE,B8,AD,01,1C, 70
520 DATAC9,37,F0,0A,E8,D0,EA,C8,D0,E7,A9,03,F0,19,8E, 94
530 DATAFE,05,8C,FF,05,C0,0B,90,09,C0,0E,B0,05,A9,01, 36
540 DATA4C,69,F9,C6,37,D0,B3,A9,03,D0,F5,20,13,05,20, 247
550 DATA42,D0,A9,24,85,0A,A9,E0,85,02,A5,02,30,FC,60, 177
560 DATAA9,05,85,37,AD,00,1C,29,9F,09,08,8D,00,1C,AD, 98
570 DATA00,1C,30,FB,B8,AD,01,1C,A2,00,50,FE,B8,AD,01, 31
580 DATA1C,C9,37,D0,EB,A2,00,A0,00,AD,00,1C,30,FB,B8, 197
590 DATAAD,01,1C,50,FE,B8,AD,01,1C,C9,37,F0,0A,E8,D0, 76
600 DATAEA,C8,D0,E7,A9,00,F0,19,8E,FE,06,8C,FF,06,E0, 30
610 DATA31,D0,0B,C0,0C,D0,07,A9,FF,85,10,4C,9E,FD,C6, 153
620 DATA37,D0,B1,A9,00,F0,F3,00,00,00,00,00,00,00,00, 68
```

Die in den Datazeilen stehenden Daten entsprechen den Maschinenspracheteilen (Auftragungs- und Abfrageprogramm), die zuerst in den Speicherbereich ab $C000 des C64 geschrieben werden, von wo sie nach der Übergabe der Parameter in die Floppy geschrieben und dort gestartet werden. Auf die einzelnen Maschinenspracheprogramme werden wir später noch eingehen. Gleich im Anschluß an dieses Programm stellen wir Ihnen das entsprechende Abfrageprogramm vor.

```
0 PRINTCHR$(147)
5 INPUT"TRACK,SEKTOR DES ABFRAGEBLOCKS ";T,S
10 OPEN1,8,15:OPEN2,8,2,"#2"
20 PRINT#1,"U1 2 0";STR$(T);STR$(S)
30 PRINT#1,"M-E"CHR$(3)CHR$(5)
40 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
50 GET#1,A$:IFASC(A$+CHR$(0))<>255THENPRINT"  SCHUTZ ERROR":END
60 PRINT" SCHUTZ OK"
```

Der einzugebende Track und Sektor richtet sich nach der Eingabe, die Sie beim Erstellen des Schutzes gemacht haben.

Für diejenigen, die sich noch näher mit diesem Schutz befassen wollen, folgen nun die zugehörigen Assemblerlistings zum Auf-

tragen und Abfragen desselben. Beide Programme werden ab $0503 gestartet. Als erstes zeigen wir Ihnen das Programm zum Auftragen des Schutzes.

```
0500 JSR $0513   'JMP' zum Überspringen des Hauptprogramms
```

Das Programm wird hier gestartet:

```
0503 JSR $D042   Disk initialisieren
0506 LDA #$24    Track-Nummer laden
0508 STA $0A     und an Job übergeben
050A LDA #$E0    Job-Code 'E0' laden
050C STA $02     und in Job-Speicher schreiben
050E LDA $02     auf Rückmeldung vom Job warten
0510 BMI $050E   verzweige, wenn keine Rückmeldung
0512 RTS         Rücksprung

0513 LDA $1C00   Control-Port laden
0516 AND #$9F    Speed auf Null stellen
0518 ORA #$08    und LED anschalten
051A STA $1C00   Wert speichern
051D JSR $FE0E   Track mit $55 löschen
0520 LDY #$10    Zähler für Anzahl der zu schreibenden
0522 LDX #$00    $27, $FF Blöcke auf 4096 stellen
0524 LDA #$27    erstes zu schreibendes Byte laden
0526 STA $1C01   und zum R/W-Kopf schicken
0529 BVC $0529   warten, bis Byte geschrieben
052B CLV         Byte-Ready-Leitung löschen
052C LDA #$FF    zweites zu schreibendes Byte laden
052E STA $1C01   und zum R/W-Kopf schicken
0531 BVC $0531   warten, bis Byte geschrieben
0533 CLV         Byte-Ready-Leitung löschen
0534 DEX         Zähler verringern
0535 BNE $0524   verzweige, wenn kein Unterlauf
0537 DEY         HIGH-Byte verringern
0538 BNE $0524   verzweige, wenn noch Bytes zu schreiben
053A LDA #$FF    $FF (SYNC-Byte) laden
053C STA $1C01   und zum R/W-Kopf schicken
053F BVC $053F   warten, bis Byte geschrieben
```

```
0541 CLV          Byte-Ready-Leitung löschen
0542 BVC $0542    warten, bis Byte geschrieben
0544 CLV          Byte-Ready-Leitung löschen
0545 BVC $0545    warten, bis Byte geschrieben
0547 CLV          Byte-Ready-Leitung löschen
0548 LDA #$33     Markierungs-Byte laden
054A STA $1C01    und zum R/W-Kopf schicken
054D BVC $054D    warten, bis Byte geschrieben
054F CLV          Byte-Ready-Leitung löschen
```

Hier beginnt der Überprüfungsteil der zuvor geschriebenen Spur:

```
0550 JSR $FE00    R/W-Kopf auf Lesen schalten
0553 LDA $1C00    Control-Port laden
0556 BMI $0553    warten, bis SYNC-Signal anliegt
0558 CLV          Byte-Ready-Leitung löschen
0559 LDA $1C01    Port freimachen
055C NOP
055D NOP
055E BVC $055E    warten, bis Byte gelesen
0560 CLV          Byte-Ready-Leitung löschen
0561 LDA $1C01    Byte vom Port holen
0564 CMP #$27     mit Normalwert vergleichen
0566 BEQ $0553    weitersuchen, bis Markierung gefunden
0568 STA $0580    Markierungs-Byte an die
056B STA $0597    entsprechenden Stellen schreiben
056E LDA $1C00    Control-Port laden
0571 BMI $056E    warten, bis SYNC-Signal anliegt
0573 CLV          Byte-Ready-Leitung löschen
0574 LDA $1C01    Port freimachen
0577 NOP
0578 NOP
0579 BVC $0579    warten, bis Byte anliegt
057B CLV          Byte-Ready-Leitung löschen
057C LDA $1C01    Byte vom Port holen
057F CMP #$37     mit Markierungs-Byte vergleichen
0581 BNE $056E    weitersuchen, wenn kein Markierungs-Byte
0583 LDX #$00     Zähler für Anzahl der SYNCs auf
0585 LDY #$00     Null setzen
```

```
0587 LDA $1C00   Control-Port laden
058A BMI $0587   warten, bis SYNC-Signal anliegt
058C CLV         Byte-Ready-Leitung löschen
058D LDA $1C01   Port freimachen
0590 BVC $0590   warten, bis Byte gelesen
0592 CLV         Byte-Ready-Leitung löschen
0593 LDA $1C01   Byte vom Port holen
0596 CMP #$37    mit Markierungs-Byte vergleichen
0598 BEQ $05A4   weitersuchen, bis Markierungs-Byte gefunden
059A INX         Zähler für SYNCs erhöhen
059B BNE $0587   verzweige, wenn kein Überlauf
059D INY         HIGH-Byte des Zählers erhöhen
059E BNE $0587   verzweige, wenn kein Überlauf
05A0 LDA #$03    sonst Fehlermeldung laden
05A2 BEQ $05BD   und übergeben, Rücksprung

05A4 STX $05FE   Zählerwerte
05A7 STY $05FF   speichern
05AA CPY #$0B    Zähler auf Zulässigkeit prüfen
05AC BCC $05B7   erneuter Schreibversuch, wenn falscher Wert
05AE CPY #$0E    Zähler auf Zulässigkeit prüfen
05B0 BCS $05B7   erneuter Versuch, wenn nicht zulässig
05B2 LDA #$01    positive Rückmeldung laden
05B4 JMP $F969   und übergeben, Rücksprung
05B7 DEC $37     Zähler für Fehlversuche verringern
05B9 BNE $056E   verzweige, wenn noch Versuche erlaubt,
05BB LDA #$03    sonst negative Rückmeldung laden
05BD BNE $05B4   und Übergeben, Rücksprung
```

Nach dem Auftragungsprogramm nun das Abfrageprogramm, welches auch ab $0503 im Speicher der Floppy gestartet wird:

```
0500 JSR $0513   'JMP' zum Überspringen des Hauptprogramms
```

Das Programm wird hier gestartet:

```
0503 JSR $D042   Disk initialisieren
0506 LDA #$24    Track-Nummer laden
0508 STA $0A     und an Job Übergeben
050A LDA #$E0    JOB-Code 'E0' laden
050C STA $02     und in Job-Speicher schreiben
050E LDA $02     warten, bis Rückmeldung
0510 BMI $050E   vom Job erfolgt
0512 RTS         Rücksprung
0513 LDA #$05    Zähler für Anzahl der Fehlversuche laden
0515 STA $37     und Wert speichern
0517 LDA $1C00   Control-Port laden
051A AND #$9F    Speed auf Null stellen
051C ORA #$08    und LED anschalten
051E STA $1C00   Wert speichern
0521 LDA $1C00   Control-Port laden
0524 BMI $0521   warten, bis SYNC-Signal anliegt
0526 CLV         Byte-Ready-Leitung löschen
0527 LDA $1C01   Port wieder freimachen
052A NOP
052B NOP
052C BVC $052C   warten, bis Byte anliegt
052E CLV         Byte-Ready-Leitung löschen
052F LDA $1C01   Byte vom Port holen
0532 CMP #$37    vergleiche, ob Markierungs-Byte gelesen
0534 BNE $0521   erneut lesen, wenn nicht gefunden
0536 LDX #$00    Zähler für Anzahl der SYNC-Markierungen
0538 LDY #$00    auf Null stellen
053A LDA $1C00   Control-Port laden
053D BMI $053A   warten, bis SYNC-Signal anliegt
053F CLV         Byte-Ready-Leitung löschen
0540 LDA $1C01   Port freimachen
0543 BVC $0543   warten, bis Byte gelesen
0545 CLV         Byte-Ready-Leitung löschen
0546 LDA $1C01   Byte vom Port holen
0549 CMP #$37    mit Markierungs-Byte vergleichen
054B BEQ $0557   verzweige, wenn gefunden
054D INX         Zähler für Anzahl der SYNCs erhöhen
054E BNE $053A   verzweige, wenn kein Überlauf
```

```
0550 INY          HIGH-Byte des Zählers erhöhen
0551 BNE $053A    verzweige, wenn kein Überlauf
0553 LDA #$00     sonst negative Rückmeldung laden
0555 BEQ $0570    unbedingter Sprung
0557 STX $06FE    Zählerwerte
055A STY $06FF    speichern
055D CPX #$31     Zähler auf Richtigkeit prüfen
055F BNE $056C    verzweige, wenn Wert falsch
0561 CPY #$0C     HIGH-Byte des Zählers überprüfen
0563 BNE $056C    verzweige, wenn Wert falsch
0565 LDA #$FF     positive Rückmeldung laden
0567 STA $10      und Wert übergeben
0569 JMP $FD9E    Rücksprung
056C DEC $37      Zähler für Fehler verringern
056E BNE $0521    erneuter Versuch, wenn noch erlaubt
0570 LDA #$00     sonst negative Rückmeldung laden
0572 BEQ $0567    unbedingter Sprung
```

6.6.4 Daten ohne SYNC-Markierung

In diesem Kapitel soll ein Kopierschutz besprochen werden, der mit gutem Erfolg professionell eingesetzt wird. Er basiert auf der Tatsache, daß Daten ohne eine SYNC-Markierung auf einen Track geschrieben, aber trotzdem ohne weiteres abgefragt werden können. Dieses System ist ohne aufwendige Programme zu verwirklichen.

Wahrscheinlich werden Sie jetzt verständnislos mit dem Kopf schütteln, denn ohne SYNC-Markierung läuft bei der Floppy-Programmierung normalerweise nichts.

Kommen wir jetzt zur Erklärung des Systems. Ein Track wird von allen SYNC-Markierungen befreit und vollständig mit einer bestimmten Byte-Kombination beschrieben. Beginnt man nun, die Bytes auf dem Track zu lesen, findet man irgendwann die gewünschte Byte-Kombination. Das Ganze hat jedoch einen Haken, denn nach diesem System kann es sehr lange dauern, bis die richtigen Daten gefunden werden. Warum dies so ist, läßt sich an einem Beispiel zeigen.

Stellen wir uns vor, wir hätten einen Track vollständig mit dem Byte $33 beschrieben, dessen Bitkombination wie folgt aussieht:

```
$33 $33 $33 = 00110011 00110011 00110011
```

Wenn wir anfangen, Daten von diesem Track zu lesen, ist jedoch nicht gewährleistet, daß die ersten Bits, die der Controller registiert, auch wirklich zwei Null-Bits sind. Sollte das nicht der Fall sein, d.h. findet er beispielsweise zuerst die Bits 0 und 1, dann wird immer anstelle der $33 eine $66 gefunden. Genauso ist es möglich, daß $CC oder $99 erkannt wird.

Da es beim Lesen von Daten immer wieder dazu kommt, daß Bits "verschluckt" werden, falls zuvor der Controller nicht mit einer SYNC-Markierung synchronisiert wurde, kommt es somit auch zu einer Veränderung der Daten, die auf dem Track gefunden werden, so daß auch irgendwann das Byte $33 gefunden wird, was aber unter Umständen erst recht spät geschieht.

Sie werden sich sicher vorstellen können, daß das Auffinden einer ganzen Byte-Kombination noch zeitaufwendiger ist. Es kann vorkommen, daß eine Byte-Kombination von drei Bytes erst nach 50000 oder mehr Leseversuchen gefunden wird. Aus diesem Grund sind wir bei unserem Schutz andere Wege gegangen.

Wir haben nach mehrmaligem Schreiben unserer Byte-Kombination die Bits der Daten um ein Bit verschoben und das letzte Bit am Ende wieder angehängt. Übertragen auf das Beispiel mit der $33 bedeutet das, daß wir den Track mit $33, $66, $CC und $99 beschreiben.

```
$33 = 00110011
$66 = 01100110
$CC = 11001100
$66 = 10011001
```

Wenn wir nach diesem Prinzip vorgehen, muß der Controller, unabhängig davon, welche Bits er zuerst gefunden hat, sehr schnell unsere gewünschten Daten finden.

Unser Kopierschutzerstellungsprogramm befreit einen Track von allen SYNC-Markierungen und schreibt 42mal die Bytefolge $2F, $53, $77 auf den Track. Daraufhin werden die Bits der drei Bytes, wie geschildert, um ein Bit verschoben und die neuentstandenen Bytes erneut 42mal geschrieben. Diese Prozedur wiederholt sich so lange, bis jede mögliche Bit-Kombination auf Disk geschrieben wurde.

Im folgenden zeigen wir Ihnen das entsprechende Programm, das einen solchen Kopierschutz erzeugt. Das Programm wird ab $054B in der Floppy gestartet.

```
0500 LDA $1C00   Control-Port laden
0503 AND #$9F    Speed auf Null stellen
0505 ORA #$08    LED anschalten
0507 STA $1C00   Wert speichern
050A JSR $FE0E   Track mit $55 beschreiben
050D LDA #$2F    erstes zu schreibendes Byte
050F STA $06     speichern
0511 LDA #$53    zweites Byte
0513 STA $07     speichern
0515 LDA #$77    drittes Byte
0517 STA $08     speichern
0519 LDX #$30    Zähler für Anzahl der Byte-Kombinationen
051B LDY #$2A    Zähler für Anzahl der gleichen Byte-Folgen
051D LDA $06     ersten Wert laden
051F STA $1C01   und zum R/W-Kopf schicken
0522 BVC $0522   warten, bis Byte geschrieben
0524 CLV         Byte-Ready-Leitung löschen
0525 LDA $07     zweites Byte holen
0527 STA $1C01   und zum R/W-Kopf schicken
052A BVC $052A   warten, bis Byte geschrieben
052C CLV         Byte-Ready-Leitung löschen
052D LDA $08     drittes Byte laden
052F STA $1C01   und zum R/W-Kopf schicken
```

```
0532 BVC $0532   warten, bis Byte geschrieben
0534 CLV         Byte-Ready-Leitung löschen
0535 DEY         Zähler verringern
0536 BNE $051D   verzweige, wenn noch Bytes zu schreiben
0538 ASL $08     zu schreibende Werte um
053A ROL $07     ein Bit
053C ROL $06     nach links schieben
053E BCC $0542   verzweige, wenn letztes Bit gelöscht war
0540 INC $08     sonst Bit vorne wieder einfügen
0542 DEX         Zähler verringern
0543 BNE $051B   verzweige, wenn noch Bytes zu schreiben
0545 JSR $FE00   R/W-Kopf auf Schreiben schalten
0548 JMP $FD9E   Rücksprung
```

Das Programm wird hier gestartet:

```
054B LDA #$24    Track-Nummer laden
054D STA $0A     und übergeben
054F LDA #$E0    Job-Code 'E0' laden
0551 STA $02     und in Job-Speicher schreiben
0553 LDA $02     warte, bis Rückmeldung erhalten
0555 BMI $0553   verzweige, wenn noch keine Rückmeldung
0557 JMP $D042   Disk initialisieren, Rücksprung
```

Im Anschluß an das Assembler-Listing folgt nun der dazugehörige BASIC-Loader, der eine Eingabe des Tracks erlaubt, auf den der Schutz aufgetragen werden soll:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 9205 THEN PRINT"ERROR IN DATAS":STOP
110 INPUT "TRACK-NUMMER";T
120 PRINT#1,"M-W"CHR$(76)CHR$(5)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(75)CHR$(5)
```

```
150 CLOSE1
302 DATA173,  0, 28, 41,159,  9,  8,141,  0, 28, 32, 14,254,169, 47,133,
  6,169
304 DATA 83,133,  7,169,119,133,  8,162, 48,160, 42,165,  6,141,  1, 28,
 80,254
306 DATA184,165,  7,141,  1, 28, 80,254,184,165,  8,141,  1, 28, 80,254,
184,136
308 DATA208,229,  6,  8, 38,  7, 38,  6,144,  2,230,  8,202,208,214, 32,
  0,254
310 DATA 76,158,253,169, 36,133, 10,169,224,133,  2,165,  2, 48,252, 76,
 66,208
312 DATA -1
```

Unser Abfrageprogramm arbeitet wie bereits beschrieben. Es fängt an, Daten auf dem Track zu lesen, und vergleicht diese auf ihre Richtigkeit. Sollte nach ca. einer Umdrehung noch nicht die richtige Byte-Kombination gefunden worden sein, so wird eine negative Rückmeldung an den Computer übergeben. Anderenfalls wird noch überprüft, ob sich auf dem Track eine SYNC-Markierung befindet. Beim Auffinden einer solchen wird ebenfalls eine negative Rückmeldung übergeben.

Als nächstes zeigen wir Ihnen das dazugehörige Abfrageprogramm, das ab $0644 gestartet wird.

```
0600 LDA $1C00    Control-Port laden
0603 AND #$9F     Speed auf Null stellen
0605 ORA #$08     LED anschalten
0607 STA $1C00    Wert speichern
060A LDX #$20     Zähler für Fehlversuche
060C LDA #$00     LOW-Byte des Zählers
060E STA $11      setzen
0610 LDY #$02     Zeiger auf zu vergleichendes Byte
0612 BVC $0612    warten, bis Byte gelesen
0614 CLV          Byte-Ready-Leitung löschen
0615 LDA $1C01    Byte vom Port laden
0618 CMP $0653,Y  mit Byte aus Tabelle vergleichen
```

```
061B BNE $0639   verzweige, wenn ungleich
061D DEY         sonst Zeiger auf Byte verändern
061E BPL $0612   warten, bis alle Bytes verglichen
```

Im folgendem Programmteil wird die Existenz einer SYNC-Markierung überprüft.

```
0620 LDX #$20    Zähler für Länge des Tracks
0622 LDY #$00    LOW-Byte des Zählers
0624 LDA $1C00   Control-Port laden
0627 BPL $0640   Fehler, wenn SYNC gefunden
0629 NOP
062A NOP
062B NOP
062C DEY         Zähler verringern
062D BNE $0624   verzweige, wenn kein Unterlauf
062F DEX         HIGH-Byte verringern
0630 BNE $0624   verzweige, wenn noch nicht fertig
0632 LDA #$FF    positive Rückmeldung laden
0634 STA $11     und übergeben
0636 JMP $FD9E   Rücksprung
0639 DEC $11     Zähler für Fehler verringern
063B BNE $0613   verzweige, wenn noch weitere Versuche
063D DEX         HIGH-Byte des Zählers verringern
063E BNE $0613   verzweige, wenn noch weitere Versuche
0640 LDA #$00    negative Rückmeldung laden
0642 BEQ $0634   unbedingter Sprung
```

Das Programm wird hier gestartet:

```
0644 LDA #$24    Track-Nummer laden
0646 STA $0C     und übergeben
0648 LDA #$E0    Job-Code 'E0' laden
064A STA $03     und in Job-Speicher schreiben
064C LDA $03     Rückmeldung abwarten
064E BMI $064C   verzweige, wenn noch keine Rückmeldung
0650 JMP $D042   Disk initialisieren, Rücksprung
```

```
0653 77 53 2F    Daten, die auf dem Track gesucht werden
```

Nun folgt wieder der dazugehörige BASIC-Loader:

```
10 OPEN1,8,15,"I"
20 READ X:IF X=-1THEN 100
25 SU=SU+X
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(X)
40 N=N+1:GOTO 20
100 IFSU <> 10459 THEN PRINT"ERROR IN DATAS":STOP
110 INPUT "TRACK-NUMMER";T
120 PRINT#1,"M-W"CHR$(69)CHR$(6)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(68)CHR$(6)
140 FORN=1 TO 500:NEXT
150 PRINT#1,"M-R"CHR$(17)CHR$(0)CHR$(1)
160 GET#1,A$
170 IF ASC(A$+CHR$(0))<>255 THENPRINT" SCHUTZ ERROR":END
180 PRINT" SCHUTZ OK"
302 DATA173,  0, 28, 41,159,  9,  8,141,  0, 28,162,128,169,  0,133, 17,
160,  2
304 DATA 80,254,184,173,  1, 28,217, 83,  6,208, 28,136, 16,242,162, 32,
160,  0
306 DATA173,  0, 28, 16, 23, 234,234,234,136,208,245,202,208,242,169,255
,133
308 DATA 17,76,158,253,198, 17,208,211,202,208,208,169,  0,240,240,169,
36
310 DATA133, 12,169,224,133,  3,165,  3, 48,252, 76, 66,208,119, 83, 47,
 -1
```

6.7 Der "Disketten-Killer"

Falls Sie nicht wissen, wie Sie eine kopierte Diskette zerstören sollen, nachdem festgestellt wurde, daß es sich um eine Kopie handelt, so haben wir hier eine kurze, aber nichtsdestoweniger wirkungsvolle Methode anzubieten.

Bei dieser Methode handelt es sich um einen BASIC-Einzeiler, der den R/W-Kopf der Floppy auf den Schreibbetrieb umschal-

tet und dadurch alle Daten löscht, über die sich der Kopf bewegt. Sämtliche Disketten, die in das Laufwerk geschoben werden, werden zerstört, sobald sich der Laufwerksmotor zu drehen beginnt. Das einzige, was Sie vor diesem "Killer" schützt, ist das Ausschalten des Laufwerks.

```
10 OPEN1,8,15:PRINT#1,"M-W"CHR$(12)CHR$(28)CHR$(1)CHR$(200) :PRINT#1,"I"
:CLOSE1
```

Achtung! Dieses Programm zerstört (mindestens) die Tracks 1 bis 18, das heißt, alle darauf befindlichen Daten sind so unwiederbringlich gelöscht, als wenn Sie die Diskette formatiert hätten.

7. Wie man sich gegen Knackmodule schützt

7.1 Einleitung

Im Zuge der in allen Bereichen der Technik um sich greifenden Rationalisierung hat man es vor einigen Monaten geschafft, den ersten Schritt zur Wegrationalisierung der "Knacker" zu tun. Damals erschienen die ersten vollautomatischen Knackmodule. Diese Module bestehen meistens aus einer in ein Gehäuse gepackten Platine mit einem Druckknopf oder Schalter. Man braucht nur das Modul in den Modulport des C64 einzustecken, dann das geschützte Programm zu laden, und nachdem dieses Programm seinen Kopierschutz abgefragt hat, den Knopf zu betätigen. Das Modul speichert dann den gesamten Speicherinhalt des Rechners, oft sogar gepackt und mit einem Schnellader versehen, auf einer Diskette oder Kassette ab, selbstverständlich ohne Kopierschutz. Wenn man das Programm wieder einlädt und startet, wird der Rechner in denselben Zustand wie zum Zeitpunkt des Knopfdruckes versetzt. Wie man sein Programm gegen solche Module sichert, soll in diesem Kapitel beschrieben werden.

7.2 Vorläufer der Knackmodule

Wie kam es überhaupt zu der Entwicklung von Knackmodulen? Viele Raubkopierer, denen es zu mühsam war, sich um einen Programmschutz herumzuwinden, gingen dazu über, mit Hilfe eines RESET-Schalters aus dem laufenden Programm herauszuspringen und die benutzten Teile des Speichers abzuspeichern. Sie brauchten dann nur noch die Startadresse herauszusuchen, und schon war die Kopie fertig. Die Programmierer, die das störte, brachten in ihren Programmen eine 'CBM80'-Kennung unter, wodurch ein RESET wirkungslos wurde. An dieser Stelle tauchten die ersten Module und Betriebssysteme auf, die speziell das "Knacken" unterstützen sollten. Mit ihnen war es möglich, die 'CBM80'-Kennung zu umgehen. Einige unterstützten sogar

das Sichern von Speicherbereichen, die normalerweise bei einem RESET grundsätzlich gelöscht wurden. Folgende Bereiche sind davon betroffen:

```
$0000-$0101 =     0-  257  Systemadressen
$0200-$0802 =   512- 2048  Systemadressen und Bildschirm
$A000       = 40960        durch RAM-Test gelöscht
$D000-$DD0F = 53248-56591  I/O-Bereich
$FD30-$FD4F = 64816-64847  beim Initialisieren der Vektoren
```

Gegen die meisten "Knack"-Betriebssysteme kann man sich schon dadurch schützen, indem man wichtige Programmteile im Bereich $0400 (=1024) bis $07FF (=2048) unterbringt. Dazu muß man aber den Bildschirmspeicher, der normalerweise ab $0400 liegt, in einen freien Teil des Speichers verschieben. Folgendes Programm legt den Bildschirmspeicher nach $CC00 und den Anfang des BASIC-Speichers nach $0400, was zwei Effekte hat: Einerseits erhält man so ein Kilobyte mehr Speicherplatz für seine Programme, andererseits wird bei einem RESET der Anfang des BASIC-Programms unwiederbringlich gelöscht.

```
100 FORI=1TO66STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
125 NEXT:READA:IFC=ATHENC=0:NEXT:SYS49152:NEW
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,A2,33,86,01,A0,00,84,FB,A9,D0,85,FC,B1,FB, 153
301 DATAE6,01,91,FB,86,01,C8,D0,F5,E6,FC,A5,FC,C9,E0, 179
302 DATAD0,ED,A9,37,85,01,58,AD,00,DD,29,FC,8D,00,DD, 148
303 DATAA9,35,8D,18,D0,A9,CC,8D,88,02,8C,00,04,C8,84, 187
304 DATA2B,A9,04,85,2C,A9,93,4C,D2,FF,A4,A4,A4,A4,A4, 22
```

Assemblerlisting:

```
C000 SEI          Interrupts sperren
C001 LDX #$33     Speicherkonfiguration umstellen:
C003 STX $01      Zeichen-ROM einschalten
```

```
C005 LDY #$00        Schleifenzähler $FB/$FC auf
C007 STY $FB         $D000 setzen
C009 LDA #$D0
C00B STA $FC
C00D LDA ($FB),Y     Byte aus Zeichen-ROM holen
C00F INC $01         RAM in $D000 einschalten
C011 STA ($FB),Y     Byte in RAM schreiben
C013 STX $01         Zeichen-ROM einschalten
C015 INY             Schleifenzähler LOW erhöhen
C016 BNE $C00D       verzweige, wenn ungleich null
C018 INC $FC         Schleifenzähler HIGH erhöhen
C01A LDA $FC         schon den
C01C CMP #$E0        Bereich $E000 erreicht?
C01E BNE $C00D       verzweige, wenn nein
C020 LDA #$37        alte Speicherkonfiguration
C022 STA $01         wieder herstellen
C024 CLI             Interrupts zulassen
C025 LDA $DD00       VIC auf
C028 AND #$FC        obersten 16K-Bereich
C02A STA $DD00       einstellen
C02D LDA #$35        Bildschirmspeicher bei $CC00 und
C02F STA $D018       Zeichengenerator bei $D000
C032 LDA #$CC        Adresse des Bildschirms dem
C034 STA $0288       Betriebssystem mitteilen
C037 STY $0400       Null an neuen BASIC-Speicher-Anfang
C03A INY             $0401 in
C03B STY $2B         den Zeiger auf den BASIC-Anfang
C03D LDA #$04        $2B/$2C schreiben
C03F STA $2C
C041 LDA #$93        $93 (=147) ist ASCII-Wert für CLR/HOME
C043 JMP $FFD2       BASOUT, hier: Bildschirm löschen
```

Am besten arbeiten Sie mit dem Programm so, daß Sie den NEW-Befehl in Zeile 125 durch einen LOAD-Befehl ersetzen, der das eigentliche Hauptprogramm in den Speicher bringt. Dieses sollte als erstes den Befehl 'CLR' ausführen, damit keine Probleme mit Variablenüberlappungen auftreten können. Außerdem darf dieses Programm ohne das vorhergehende Ladeprogramm nicht lauffähig sein, was sich am einfachsten durch die

Übergabe eines Wertes von dem ersten Programm an das zweite per POKE in eine freie Speicherstelle realisieren läßt. Das ganze könnte zum Beispiel so aussehen: Nach erfolgter Kopierschutzabfrage in dem Teil, der den DATA-Lader enthält, wird der Befehl 'POKE 2,123' benutzt. Im Hauptprogramm wird dann abgefragt, ob der Wert der Speicherstelle Zwei wirklich gleich 123 ist. Falls nein, wird das Programm abgebrochen. Es ist wohl unnötig, zu erwähnen, daß Sie daran denken, Ihr Programm gegen die Einblicke anderer zu schützen.

Gegen gute Knackmodule wird man mit dieser Methode allerdings nicht sehr weit kommen, da diese immer über ein eigenes RAM verfügen, in das sie die Speicherbereiche, die sie benötigen, hinüberretten, also auch den Bildschirmspeicher. Um sich auch hiervor zu sichern, muß man schon schwerere Geschütze auffahren.

7.2 Knackmodule neueren Datums

Alle Schutzmethoden gegen Knackmodule haben eines gemeinsam: sie enthalten eine periodisch oder auch nur an wichtigen Stellen des Programms aufgerufene Schutzabfrage, da ein nur einmalig abgefragter Kopierschutz gegen ein solches Modul sinnlos ist. Es hängt unter Umständen stark von dem Programm ab, wie man diese Abfrage installiert. Zum Beispiel könnte man in einem "Adventure" alle 100 Schritte den Spieler fragen, welches Wort sich auf einer bestimmten Seite der Anleitung an einer bestimmten Stelle befindet. Allerdings lassen sich Anleitungen ebenfalls kopieren, wenn auch meist mit größerem Aufwand als Programme. Daher gehen einige Firmen dazu über, ihren Programmen eine spezielle Linse beizufügen, ohne die es nicht möglich ist, die von Zeit zu Zeit verschlüsselt angezeigten Buchstaben zu entziffern. Solche Schutzsysteme stellen nicht nur ein Handikap gegenüber den "Knackern" dar, sondern auch eine Zumutung gegenüber dem Anwender. Außerdem animieren so geschützte Programme gerade dazu, den Schutz zu entfernen, da eine ungeschützte Kopie wesentlich anwenderfreundlicher ist.

Sinnvoller und mit Sicherheit von keinem Knackmodul zu kopieren ist das mehrfache Abfragen des auf der Diskette angebrachten Schutzes. Dies sollte am besten dann geschehen, wenn das Programm Daten nachlädt. Falls man größeren Aufwand treiben will, kann man auch ein komplett neues Diskettenformat verwenden. Man sollte aber darauf achten, daß der Anwender davon nicht betroffen wird, also seine geschützten Disketten nicht öfter wechseln muß als die ungeschützten.

Was aber, wenn Ihr Programm nach Einladen des Hauptprogramms nur noch mit einer Datendiskette arbeitet und die Originaldiskette nicht mehr benötigt? Oder was, wenn Sie Ihr Programm gar nicht auf einer Diskette unterbringen, sondern auf einer Kassette?

Betrachten wir zuerst den Fall, daß eine Diskettenstation vorhanden ist. Dann ist es am einfachsten, nach erfolgter Kopierschutzabfrage einige Bytes, wenn nicht sogar ein komplettes Programm, im Speicher der Floppy abzulegen und regelmäßig das Vorhandensein dieser Bytes abzufragen. Bei normalem Diskettenbetrieb bleiben folgende Speicherstellen ungenutzt:

Hexadezimal	Dezimal
0005	5
0010 / 0011	16 / 17
0014 / 0015	20 / 21
001B	27
001D	29
001F	31
0021	33
0023	35
0035	53
0037	55
003B / 003C	59 / 60
0046	70
0096	150
0100	256
0102 / 0103	258 / 259
02FB	763
02FD	765

Mit folgendem Befehl schreibt man ein oder mehrere Bytes in eine bestimmte Speicheradresse der Floppy:

```
OPEN 1,8,15,"M-W"+CHR$(LOW-Byte)+CHR$(HIGH-Byte)
+CHR$(Anzahl)+CHR$(Byte1)+CHR$(Byte2)+...:CLOSE 1
```

Die Bezeichnungen LOW- und HIGH-Byte beziehen sich auf die Anfangsadresse des zu belegenden Bereichs. Beispiel: Sie wollen in die Speicherstelle 258 den Wert 123 eintragen. Die Anzahl der zu sendenden Bytes beträgt eins, das LOW-Byte von 258 (=$0102) ist zwei, das HIGH-Byte eins. Der Befehl sieht dann so aus:

```
OPEN 1,8,15,"M-W"+CHR$(2)+CHR$(1)+CHR$(1)+CHR$(123):CLOSE 1
```

Um den Inhalt einer Speicherstelle wieder auszulesen, benötigt man folgenden Befehl:

```
OPEN 1,8,15,"M-R"+CHR$(LOW-Byte)+CHR$(HIGH-Byte)
GET #1,A$:A=ASC(A$+CHR$(0)):CLOSE 1
```

A enthält dann den Wert der angesprochenen Speicherstelle. Zu dem oben angegebenen Beispiel sieht die Abfrage dann so aus:

```
OPEN 1,8,15,"M-R"+CHR$(2)+CHR$(1)
GET #1,A$:A=ASC(A$+CHR$(0)):CLOSE 1
```

A muß dann den Wert 123 enthalten.

Der Schutz beruht einfach darauf, daß Knackmodule den Inhalt des Floppy-Speichers nicht sichern. Denken Sie bitte immer daran, daß das Schreiben des Bytes nur ein einziges Mal im Programm vorkommen darf, nämlich bei der Kopierschutzabfrage, und daß der Test, ob das Byte vorhanden ist, regelmäßig durchzuführen ist, am besten vor dem Aufruf oft gebrauchter Programmteile.

Was jedoch läßt sich tun, wenn Sie keine Diskettenstation zur Verfügung haben? Gibt es eine Methode, Knackmodule auszutricksen, die völlig speicherintern abläuft? Die Antwort ist: Ja, es gibt sie. Die Idee ist einfach die, daß im C64 einige Bausteine vorhanden sind, in deren Register man zwar einen Wert hineinschreiben kann, aber aus denen sich dieser Wert nicht direkt wieder auslesen läßt.

Das erste Beispiel, das wir in diesem Zusammenhang betrachten, ist der Soundchip des C64, der sog. *SID*. Dort läßt sich für jede der drei erzeugbaren Stimmen eine Hüllkurve festlegen. Dabei handelt es sich um den Lautstärkeverlauf eines gespielten Tones. Normalerweise kann man die einmal in ein Register geschriebenen Hüllkurvenwerte nicht wieder auslesen, was das Knackmodul daran hindert, diese Werte zu kopieren. Dummerweise bedeutet das aber auch, daß man selbst nicht testen kann, ob die Werte stimmen. Für Stimme Drei existiert allerdings ein Leseregister, das es ermöglicht, den Lautstärkeverlauf eines gerade gespielten Tones mitzuverfolgen. Die Abfrage der Hüllkurve geschieht also einfach durch Anspielen dieser Stimme. Für diesen Schutz benötigen wir folgende Register:

Hex	Dezimal	Funktion
$D412	54290	Steuerregister für Stimme Drei. Wird in diesem Register das Bit 1 von Null auf Eins geschaltet, so wird ein Ton gepielt.
$D413	54291	Attack/Decay: Bits 0 bis 3: Zeit, in der Lautstärke vom Maximum auf den Sustain-Pegel abfällt. Bits 4 bis 7: Zeit, in der Lautstärke von Null auf das Maximum ansteigt.
$D414	54292	Sustain/Release: Bits 0 bis 3: Zeit, in der Lautstärke vom Sustain-Pegel auf Null abfällt. Bits 4 bis 7: Sustain-Pegel: Lautstärke, die kurz nach Anspielen eines Tones erreicht wird.
$D41C	54300	augenblickliche Lautstärke der Stimme Drei

Beim Anspielen der Stimme Drei braucht man weder die Gesamtlautstärke des SIDs noch eine Wellenform für diese Stimme einzuschalten, wodurch der Ton unhörbar bleibt. Am einfachsten ist es, die Attack-, Decay- und Releasezeiten auf null zu stellen und den Sustainpegel auf einen Wert, der sich durch Auslesen von $D41C (=54300) feststellen läßt. Das Einstellen könnte dann so aussehen:

```
10 A=10:REM (änderbarer) Sustain-Wert
20 POKE54291,0:POKE54292,A*16
```

Setzt man in Zeile 10 A auf einen anderen Wert zwischen 0 und 15, erhält man dementsprechend einen anderen Sustain-Pegel. Gleiches gilt auch für die Abfrage:

```
10 A=10:REM (AENDERBRER) SUSTAIN-WERT
20 POKE 54290,0:POKE 54290,1:REM TON SPIELEN
30 FORI=1TO50:NEXT:REM ATTACK/DECAY-ZEIT ABWARTEN
40 IF INT(PEEK(54300)/16)<>A THEN PRINT "DU RAUBKOPIERER!"
```

Sie können selbstverständlich auch kompliziertere Hüllkurven programmieren und abfragen. Allerdings wird durch diesen Schutz die Stimme Drei belegt. Was aber, wenn man die Schutzabfrage zu einem Zeitpunkt benötigt, bei dem gerade diese Stimme schon anderweitig benutzt wird? Nun, es gibt noch andere Register, die sich normal nicht auslesen lassen, deren Wert man aber mit einem Trick erfahren kann: die Alarmzeiten der Echtzeituhren.

Der C64 besitzt zwei Chips, genannt CIAs, die beide eine Echtzeituhr besitzen. Die folgenden Erklärungen beziehen sich nur auf den ersten der beiden Chips, dessen Basisadresse bei $DC00 (=56320) liegt. Wenn Sie den anderen Chip benutzen wollen, so denken Sie bitte daran, daß sich dessen Basisadresse bei $DD00 (=56576) befindet.

Der Trick, mit dem man die Alarmzeiten überprüfen kann, ist einfach der, daß man die Echtzeituhr auf einen Wert knapp vor dem erwarteten Alarm setzt und kontrolliert, ob der Alarm kurz darauf ausgelöst wird. Dieser Schutz ist besonders schwer zu kopieren. Bei der Hüllkurve wäre es ja denkbar, daß das Knackmodul ebenfalls die Stimme Drei spielt und aufgrund des ausgelesenen Lautstärkeverlaufs die Werte für Attack, Decay, Sustain und Release herausfindet. Bei der Echtzeituhr müßte man aber, wenn man die Alarmzeit nicht schon kennt, die Uhr bis zu 24 Stunden laufen lassen, um die Zeit zu rekonstruieren.

Folgende Register braucht man zur Programmierung der Uhr:

Hex	Dezimal	Funktion
$DC08	56328	Bit 0 bis 3: Zehntelsekunden
		Bit 4 bis 7: müssen null sein
$DC09	56329	Bit 0 bis 3: Einersekunden
		Bit 4 bis 6: Zehnersekunden
		Bit 7: muß null sein
$DC0A	56330	Bit 0 bis 3: Einerminuten
		Bit 4 bis 6: Zehnerminuten
		Bit 7: muß null sein
$DC0B	56331	Bit 0 bis 3: Einerstunden
		Bit 4: Zehnerstunde
		Bit 5 bis 6: müssen null sein
		Bit 7: 0=vormittags (AM)
		1=nachmittags (PM)
$DC0D	56333	Bit 2: 1=Gleichheit von Uhrzeit und Alarmzeit; Achtung: beim Lesen dieses Registers werden alle Bits gelöscht.
$DC0F	56335	Bit 7: 0=Schreibzugriffe auf die vorhergehenden Register beziehen sich auf die Echtzeituhr. 1=Schreibzugriffe auf die vorhergehenden Register beziehen sich auf die Alarmzeit

Die Echtzeituhr wird gestellt, indem man Bit 7 von $DC0F auf null setzt und dann die Uhrzeit, beginnend mit der Stunde, in die Register einträgt. Die Uhr hält beim Ansprechen des Stundenregisters an und läuft erst beim Setzen der Zehntelsekunden weiter. Entsprechend sollte man beim Auslesen ebenfalls zuerst auf das Stundenregister zugreifen, da dann die Uhr scheinbar bis zum Lesen der Zehntelsekunden gestoppt wird. Intern läuft sie allerdings weiter. Die Alarmzeit wird genauso festgesetzt, nur muß Bit 7 von $DC0F den Wert eins haben.

Ein Programm, das beispielsweise die Alarmzeit 7 Uhr 35 Minuten 11 Sekunden und 1 Zehntelsekunde nachmittags setzen soll, sieht so aus:

```
10 POKE56335,PEEK(56335)OR128:REM BIT 7 setzen
20 POKE56331,128+7:REM 7 UHR NACHMITTAGS
30 POKE56330,3*16+5:REM 35 MINUTEN
40 POKE56329,1*16+1:REM 11 SEKUNDEN
50 POKE56328,1:REM 1 ZEHNTELSEKUNDE
```

Beim Test auf die Alarmzeit stellen wir die Uhr auf eine Zehntelsekunde vor der erwarteten Zeit, warten diese Zehntelsekunde ab und testen dann Bit 2 von Register $DC0D. Da dieses Register beim ersten Lesezugriff wieder gelöscht wird, müssen wir verhindern, daß das Betriebssystem nach erfolgtem Alarm noch vor uns auf $DC0D zugreift. Man erreicht das durch Abschalten des Timer-IRQs (siehe auch: CIA-Beschreibung im Teil "Kassettenkopierschutz"), indem man nach $DC0D (=56333) den Wert 1 schreibt. Hinterher muß man den Timer-IRQ durch 'POKE 56333,129' wieder einschalten. Bei der CIA zwei ($DD00) kann man sich diese Prozedur sparen, da sie nicht vom Betriebssystem benutzt wird.

Hier also das Programm, welches die oben eingetragene Alarmzeit testet:

```
10 POKE56335,PEEK(56335)AND128:Bit 7 LOESCHEN
20 POKE56331,128+7:REM 7 UHR NACHMITTAGS
30 POKE56330,3*16+5:REM 35 MINUTEN
```

```
40 POKE56329,1*16+1:REM 11 SEKUNDEN
50 POKE56328,0:REM 0 ZEHNTELSEKUNDEN
60 POKE56333,1:REM TIMER-IRQ ABSCHALTEN
70 FORI=1TO200:NEXT:REM ALARM ABWARTEN
80 A=PEEK(56333):POKE56333,129:REM ALARM TESTEN, IRQ EIN
90 IF (A AND 4)<>4 THEN PRINT"VERFLIXT, EIN KNACKMODUL!"
```

Zur Benutzung dieser Programme gilt das gleiche wie bei den vorhergehenden Beispielen. Die Alarm-Setz-Routine sollte ein einziges Mal in Verbindung mit der Kopierschutzabfrage ausgeführt werden, die Testroutine dagegen an allen wichtigen Stellen im Programm. Denken Sie daran, daß ein guter Kopierschutz nur in Verbindung mit einem mindestens genauso guten Programmschutz sinnvoll ist. Es gibt schließlich noch genügend "Knacker", die sich nicht auf ein Knackmodul verlassen. Gerade in BASIC-Programmen lassen sich besonders leicht Schutzabfragen finden. Kompilieren Sie daher nach Möglichkeit Ihr Programm, um es den Raubkopierern nicht leichter zu machen als nötig.

7.3 Knackmodule: Zukunftsaussichten

Es ist abzusehen, daß in Zukunft Knackmodule auf dem Markt erscheinen werden, die sich durch die beschriebenen speicherinternen Tricks nicht abschrecken lassen. Theoretisch könnte nämlich ein Modul den Adreß- und Datenbus des C64 überwachen und so alle Schreibzugriffe auf nicht lesbare Register zusätzlich in einem eigenen RAM nachvollziehen. Außerdem ist ein Modul-System denkbar, welches das Sichern des Floppy-Speichers ermöglicht. In beiden Fällen ist aber der notwendige Hardware-Aufwand wesentlich höher als bei den jetzigen Knackmodulen. Der sicherste Schutz ist und bleibt daher eine mehrfache Abfrage der Originaldiskette. Es bleibt dabei nur zu hoffen, daß nicht doch eines Tages das "alles kopierende" Kopierprogramm entwickelt wird.

8. Gerüchteküche

8.1 Der rechnerzerstörende Kopierschutz

Die Diskussion um Programme, die den C64 zerstören können, ist schon älter. Sie besteht mindestens seit der Zeit, als man sich daran erinnerte, daß zum Beispiel einige der früheren Personalcomputer nach Eingabe einiger bestimmter POKE-Befehle nur noch durch eine Reparatur wieder zum Laufen zu bekommen waren. Als ein Mitarbeiter der Firma DATA BECKER einer bekannten Homecomputer-Zeitschrift gegenüber äußerte, wahrscheinlich würden in Zukunft alle DATA BECKER-Programme, an denen man einen illegalen Kopierversuch unternehmen würde, dem benutzten Computer das Lebenslicht ausblasen, flammten die Gerüchte wieder auf. Natürlich war diese Drohung ohne jede Grundlage. Man stelle sich vor: Ein ehrlicher Anwender kauft sich ein Programm, das aufgrund einer Fehlbedienung oder einer ungenau justierten Diskettenstation seinen Kopierschutz nicht erkennt und deshalb dem Rechner ein jähes Ende bereitet. Der Hersteller des Programms sähe einer Flut von Schadensersatzklagen entgegen. Außerdem sind, soweit uns bekannt, noch nirgendwo rechnerzerstörende Programme veröffentlicht worden. Wir haben daher keine Risiken gescheut, um festzustellen, was wirklich an dem Gerücht "dran" ist.

Um es gleich vorwegzunehmen: Unsere Computer sind noch alle voll funktionstüchtig. Wir haben keine Möglichkeit gefunden, per Software irgendwelche Chips im C64 zu zerstören. Wir können selbstverständlich nicht dafür garantieren, daß die im folgenden beschriebenen Methoden bei jedem C64 unwirksam bleiben.

Man kann einen Computer auf elektronischem Wege an sich nur durch eine zu hohe Spannung oder einen Kurzschluß zerstören. Ersteres läßt bei unserem Gerät von vornherein ausschließen, da die vorhandenen Chips nur ein Schalten von Spannungen von null bis fünf Volt, also im für Chips harmlosen Bereich, ermöglichen. Ein Kurzschluß dagegen läßt sich theoretisch erreichen, indem man Leitungen von Ein-/Ausgabe-Chips, die an sich nur

als Eingang konzipiert sind, auf Ausgang schaltet und eine der Eingangsspannung entgegengesetzte Spannung programmiert. Dazu bieten sich der Prozessor 6510 und die beiden CIAs 6526 an. Beipielsweise lassen die von der Tastatur kommenden Anschlüsse dieses Verfahren zu. Wir haben hier alle in Frage kommenden Leitungen überprüft, ohne daß es den Chips etwas geschadet hätte. Vielleicht kann es im Dauerbetrieb oder bei einigen "empfindlicheren" Rechnern zu einem Schaden kommen. Gleiches gilt auch für die Ein-/Ausgabe-Chips der Diskettenstation.

Ein anderes Vorgehen zur Schädigung des Rechners könnte eine mechanische Überbeanspruchung des Steppermotors des Floppy-Laufwerks sein. Erfahrungsgemäß hält der Motor aber einiges aus. Außerdem kann man das harte Anschlagen des Steppers an die obere oder untere Begrenzung deutlich hören. Bevor ein zu harter Anschlag etwas ausrichten kann, wird sicherlich kein Computerbesitzer den Griff zum Ein-/Aus-Schalter scheuen.

Wir sehen daher keinen Ansatz für ein computerzerstörendes Programm. Sollten Sie vielleicht doch eine Methode finden, die wir übersehen haben, so wären wir für Ihre Mitteilung äußerst dankbar.

8.2 Viren

Vor einigen Monaten fand man in der Presse Berichte über Programme, die in große Datenbanksysteme eingespeist worden waren und Wochen später zur Löschung aller Daten führten. Diese Programme waren teilweise geschickt genug, sich selbst zu reduplizieren und sich an andere Programme anzuhängen, weshalb man ihnen den Namen "Viren" gab. Es war natürlich naheliegend zu fragen, ob sich auch ein Virus für den C64 konstruieren läßt. Ein solches Virus sollte sich in alle Programme auf einer Diskette einbinden. Wenn eines dieser Programme auf eine andere Diskette kopiert wird, kann sich das Virus weitervervielfältigen. Weiterhin könnte mitgezählt werden, die wievielte Kopie von einem Programm angefertigt wurde beziehungsweise

wie oft das Programm schon eingeladen wurde. Würde hier eine Höchstzahl überschritten werden, so sollte das Virus alle Daten der aktuellen Diskette zerstören.

Theoretisch ist es ohne weiteres denkbar, ein Virus zu erstellen. Dazu schreibt man sich ein Programm, das schon während des Ladens einen Autostart ausführt. Dieses Programm kopiert dann die Blöcke, in denen es sich selbst befindet, in freie Blöcke hinein und stellt die Blockverbindungszeiger so um, daß alle anderen Programme ebenfalls das Virus enthalten. Dadurch wird das Virus beim Kopieren mit einem Kopierprogramm auf eine neue Diskette mitübertragen.

Praktisch gesehen hat ein Virus aber keinen Wert. Damit es sich ausbreiten kann, muß es unbedingt unauffällig sein. Das bedeutet aber, daß sich ein Programm mit Virus zumindest am Bildschirm genauso verhalten muß wie ohne es. Daraus folgt jedoch, daß der Anwender das Programm nach dem Einladen mit dem Befehl SAVE wieder abspeichern kann, wodurch das Virus wieder entfernt wird. Weiterhin wird sich wohl jeder Computerbesitzer wundern, wenn alle Programme auf einer Diskette plötzlich vier Blöcke mehr benötigen oder wenn ein Programm beim Einladen unnatürlich viele Stepperbewegungen ausführt. Viele 64er-Besitzer haben auch Betriebssysteme, die die Startadresse des geladenen Programms anzeigen, wodurch das Virus sofort zu erkennen ist. Hat man einen Schreibschutz auf seinen Disketten, so funktioniert das Virus sowieso nicht. Zudem können Viren den Ablauf von nachladenden Programmen oder die Verwendung eines Schnelladeprogramms stören.

Weder die Raubkopierer noch die ehrlichen Anwender brauchen sich deshalb Sorgen zu machen, ein Virusprogramm könnte ihre Disketten zerstören. Wer absolut kein Risiko eingehen will, sollte bei allen selbststartenden oder nachladenden Programmen einen Schreibschutz verwenden.

9. Allgemeine Tips

Neben den speziellen Erklärungen und Hinweisen in den einzelnen Kapiteln wollen wir Ihnen hier einige allgemeine Tips im bezug auf den Programmschutz geben.

Als erstes müssen Sie darauf achten, daß von der Kopierschutzabfrage bis zum Directory alles codiert oder durch andere Methoden geschützt wird. Zuerst muß also die Kopierschutzabfrage selbst geschützt werden. Hier wird es von Vorteil sein, die Abfrage nicht zu offensichtlich in dem Programm stehen zu lassen, damit sie nicht zu einfach (zum Beispiel mit einem Monitor) auszubauen ist. Die Diskettenzugriffsbefehle, zum Beispiel B-E, sollten nach Möglichkeit in einer verschlüsselten Form vorliegen. Noch besser ist es, das Programm mittels M-W in die Floppy zu schicken und es dort zu starten.

Danach sollte das Programm mit dem Kopierschutz codiert werden, um auch hier den Zugriff zu erschweren. Die Decodierroutine kann dann ihrerseits durch eine Routine, die wesentlich kürzer und unauffälliger ist, gesichert werden.

Schließlich kommt es auch auf den Schutz des Programm-Files an. Das File kann durch diverse Directory-Manipulationen und vor allem durch einen guten Autostart gesichert werden. Durch diese Kombination der verschiedenen Schutzsysteme wird ein Fremdzugriff erheblich erschwert.

Darüber hinaus existieren eine Reihe weiterer Tricks, die vor allem darauf abzielen, jeden unbefugten Benutzer zu verwirren und den Einsatz von Hilfsprogrammen, wie Monitore und Single-Step-Simulatoren, zu verhindern.

Allgemein sollte bei der Benutzung der Betriebssystemroutinen nicht über die Sprungtabelle am Ende des ROMs verzweigt werden, da diese Adressen zu einprägsam sind. Diese Routinen können besser direkt oder über indirekte Sprünge aufgerufen werden. Eine weitere Alternative bieten selbstmodifizierende Pro-

gramme. Dabei wird die Zieladresse des JMP-Befehls erst während des Programmablaufs auf ihren endgültigen Wert gesetzt.

Als zweckmäßig hat sich ein Prinzip erwiesen, bei dem der untere Bereich des Stapels mit den Adressen der Unterprogramme vollgeschrieben oder überladen wird. In der ersten Unterroutine wird der Stapelzeiger dann umgestellt und die Routine mit einem RTS-Befehl abgeschlossen. Durch den umgesetzten Stapelzeiger wird nun das nächste Unterprogramm aufgerufen. Dieser Vorgang kann beliebig oft wiederholt werden, wenn der Zeiger umgesetzt wird.

Die Verwendung von Illegal-Opcodes zerstört die Programmstruktur, was teilweise von Vorteil sein kann, wie in Kapitel 3 beschrieben wurde.

Um den Einsatz von Hilfsprogrammen, zum Beispiel Monitore und Single-Step-Simulatoren, zu verhindern, sollte das Programm in möglichst viele Speicherbereiche zerstreut werden. So kann kein Programm geladen werden, ohne dabei einen Teil des eigentlichen Programms zu zerstören. Hierbei sollten die Speicherbereiche unter dem ROM und der Bildschirmspeicher benutzt werden. Außerdem sollten die einzelnen Speicherbereiche nach dem Abarbeiten der Routinen, die nicht mehr benötigt werden, zerstört werden.

Um die Single-Step-Simulatoren "unschädlich" zu machen, sollten Sie folgende Punkte beachten:

- durch den SEI-Befehl den Interrupt maskieren,
- den IRQ- und NMI-Vektor öfter initialisieren, das heißt, die Vektoren in der Zeropage auf ihre Originaladressen zurücksetzen.

Die oben aufgezählten Tips sollen Ihnen Hinweise zum Schutz Ihrer eigenen Programme geben. Genauere Erläuterungen finden Sie in den einzelnen Kapiteln dieses Buches.

10. Komplettlösung

Da in diesem Buch sehr viele Kopier- und Programmschutzanwendungen beschrieben werden, ist es unerläßlich, einige Beispiele dafür zu liefern, welche verschiedenen Schritte man durchzuführen hat, bis ein Programm "gründlich" geschützt ist.

Schutz eines BASIC-Programms, welches mit der Datasette aufgezeichnet werden soll:

Folgendes Programm bietet dem Benutzer drei Menüpunkte an, die nichts weiter machen, als den Text "MENUEPUNKT X" anzuzeigen. Es könnte somit das Gerüst eines umfangreicheren Programmes darstellen.

```
100 PRINT CHR$(147)
110 FOR I=1 TO 6:PRINT:NEXT
120 PRINT" 1. MENUEPUNKT A":PRINT
130 PRINT" 2. MENUEPUNKT B":PRINT
140 PRINT" 3. MENUEPUNKT C":PRINT
150 PRINT"BITTE WAEHLEN SIE AUS!"
160 GET A$
170 IF A$<"1" OR A$>"3" THEN 160
180 ON VAL(A$) GOSUB 300,400,500
190 GOTO 100
300 PRINT CHR$(147);"MENUEPUNKT A":GOTO 600
400 PRINT CHR$(147);"MENUEPUNKT B":GOTO 600
500 PRINT CHR$(147);"MENUEPUNKT C":GOTO 600
600 GET A$:IF A$="" THEN 600
610 RETURN
```

Gegen das Betätigen von RUN/STOP oder RUN/STOP-RESTORE braucht das Programm nicht geschützt werden, da dieser Schutz schon durch unseren Kassetten-Schnellader erzeugt wird. Um es auch gegen einen RESET-Schalter (siehe Kapitel 2.3) zu sichern, ergänzen Sie es bitte um folgende Zeilen:

```
20 POKE 56,128:CLR
30 POKE32768,226:POKE32769,252:POKE32770,226
40 POKE32771,252:POKE32772,195:POKE32773,194
50 POKE32774,205:POKE32775, 56:POKE32776, 48
```

Da wir das Programm hinterher mit einem Autostart versehen werden, werden diese Befehle sofort nach dem Einladen ausgeführt, wodurch sie überhaupt erst sinnvoll werden.

Selbstverständlich darf unser Programm nicht von einem Knackmodul zu kopieren sein. Daher verwenden wir eine Abfrage der Alarmzeit einer Echtzeituhr (siehe Kapitel 7.2). Zuerst muß die Alarmzeit gesetzt werden:

```
60 POKE56335,PEEK(56335)OR128
65 POKE56331,3
70 POKE56330,35
75 POKE56329,84
80 POKE56328,8
```

Die hier verwendete Uhrzeit ist drei Uhr vormittags, 25 Minuten, 54 Sekunden und acht Zehntelsekunden.

Die eigentliche Abfrage wird als Unterprogramm ab Zeile 700 installiert. Sie muß in regelmäßigen Abständen aufgerufen werden, am besten vor der Ausführung jedes Menüunterpunktes:

```
175 GOSUB 700
```

Hier nun die eigentliche Abfrage:

```
700 POKE56335,PEEK(56335)AND127
710 POKE56331,3
720 POKE56330,35
730 POKE56329,84
740 POKE56328,7
750 POKE56333,1
760 FOR I=1TO200:NEXT
```

```
770 A=PEEK(56333):POKE56333,129
780 IF (A AND 4)<>4 THEN SYS 64738
790 RETURN
```

Nachdem nun alle Änderungen im Programm stattgefunden haben, können wir es jetzt vor den neugierigen Blicken anderer und vor eventuellen Änderungen sichern. Sie speichern dazu das Programm erst einmal ab, laden dann das Packer-Programm aus Kapitel 2.1.4 und starten es mit 'RUN'. Laden Sie als nächstes wieder das Beispielprogramm und schützen Sie es durch 'SYS 49152' und 'SYS 49730'. Speichern Sie bitte auch diese Version noch einmal ab.

Um die endgültige Aufnahme vorzubereiten, laden Sie dann das Programm "KKS" aus Kapitel 5.5.3 und starten es mit 'RUN' und 'SYS 49152'. Danach holen Sie zum dritten Mal das Beispiel in den Speicher und nehmen es mit dem KKS auf.

Das Programm ist nun rundum geschützt. Man kann es mit keinem Kopierprogramm kopierens und es läßt sich auch nicht nach dem Einladen ohne weiteres unterbrechen. Selbst ein RESET-Schalter oder ein Knackmodul hilft einem nicht weiter. Sicherlich können Sie den Schutz noch weiter verbessern. Hier sind Ihrer Kreativität keine Grenzen gesetzt.

Komplettlösung für Diskette

Im folgenden Beispiel wollen wir Ihnen zeigen, wie Sie mit Hilfe der von uns gezeigten Schutzsysteme ein Maschinenspracheprogramm mit zusätzlicher Kopierschutzabfrage wirkungsvoll schützen können.

Bevor Sie sich an den Computer setzen und beginnen, Ihr Programm zu schützen, müssen Sie sich natürlich überlegen, wie Ihr Programm geschützt werden soll.

In diesem Beispiel wollen wir unser Programm als erstes codieren, und unsere Wahl fiel auf den Single-Step-Codierer aus Kapitel 3.3.4, da diese Codierung von einem Knacker nur sehr

schwer nachzuvollziehen ist. Der Nachteil bei dieser Codierung ist jedoch die drastische Verringerung der Ablaufgeschwindigkeit unseres geschützen Programms. Aus diesem Grund haben wir nicht das gesamte Programm mit dieser Codierung versehen, sondern wir haben eine andere, wesentlich kürzere Entcodierschleife benutzt. Diese Schleife entcodiert unser restliches Programm. Für diese zweite Codierung wird jedes Programmbyte mit 'EOR #$45' verknüpft und daraufhin um eins verringert.

Weil wir unser Programm auch dagegen schützen wollten, daß man es mit 'RESET' abbricht und sich das entcodierte Programm ansehen kann, haben wir es in das Bildschirm-RAM kopiert und den Bildschirm auf $0800 (2048) verlegt. Bei einem 'RESET' wird so das Programm unweigerlich zerstört.

Wir wollten verhindern, daß ein Knacker eine CBM80-Erkennung benutzt und dann beim Betätigen eines 'RESET' zu seiner eigenen Routine verzweigt, mit der er das Bildschirm-RAM rauskopiert und sich somit trotzdem unser Programm ansehen kann. Deshalb haben wir selber eine CBM80-Erkennung geschrieben, mit der wir die 'RESTORE-Taste' sperren und einen 'RESET' unbrauchbar machen.

Selbstverständlich haben wir nicht auf eine Kopierschutzabfrage verzichtet. In unserem Beispiel verwenden wir den Kopierschutz aus Kapitel 6.6.3, der einen Track mit über 3000 SYNC-Markierungen beschreibt.

Um die Reihe der Schutzsysteme noch zu erweitern, fragen wir die Rückmeldung der Kopierschutzabfrage immer wieder ab und erhalten so einen sicheren Schutz gegen Knackmodule, denn ein Knackmodul kann unmöglich auch das RAM der Floppy speichern.

Als erstes zeigen wir Ihnen das Programm, das wir schützen wollen. Die Single-Step-Entcodierroutine muß in unserem Beispiel ab $091E im Speicher stehen, von wo sie nach $C000 kopiert und gestartet wird:

```
0810 LDX #$00     X-Register löschen
0812 LDA $091E,X  Entcodierroutine
0815 STA $C000,X  nach $C000
0818 INX          schreiben
0819 BNE $0812    verzweige, wenn noch Bytes zu kopieren
081B JSR $C000    Routine aufrufen
081E NOP
081F NOP
0820 NOP
```

Der folgende Programmteil muß vor dem Start des Programms schon codiert worden sein:

```
0821 LDX #$00     Zähler für Entcodierung setzen
0823 INC $083B,X  jedes Byte um eins erhöhen
0826 INX          Zähler erhöhen
0827 BNE $0823    verzweige, wenn nicht fertig
0829 LDA $083B,X  Byte laden
082C EOR #$45     entcodieren
082E STA $043B,X  und in den Bildschirm schreiben
0831 INX          Zähler erhöhen
0832 BNE $0829    verzweige, wenn noch nicht fertig
0834 SEI          Step-Codierung unterbrechen
0835 JSR $FF84    und ganz
0838 JSR $FF8A    beenden
083B JMP $043E    zum Programm im Bildschirm springen
```

Der folgende Programmteil liegt bei der Ausführung des Programms im Bildschirmbereich ab $043E:

```
083E LDX #$0C     Zähler setzen
0840 LDA $0511,X  CMB80-Erkennung
0843 STA $8000,X  nach $8000 schreiben
0846 DEX          Zähler verringern
0847 BPL $0840    verzweige, wenn noch Daten zu schreiben
0849 LDA $D018    Bildschirm
084C AND #$0F     nach $0800
084E ORA #$20     verlegen
0850 STA $D018
```

```
0853 LDA #$08     und dem Betriebssystem mitteilen
0855 STA $0288    wo er sich jetzt befindet
0858 JSR $E544    Bildschirm löschen
085B JSR $0467    Kopierschutz abfragen
085E INC $D020    Rahmenfarbe erhöhen
0861 JSR $04F2    auf Knackmodul testen
0864 JMP $045E    in Endlosschleife laufen

0867 JSR $04CD    OPEN1,8,15
086A JSR $04DE    OPEN2,8,2,"#2"
086D LDX #$01     Ausgabe auf Kanal
086F JSR $FFC9    1 legen (CMD1)
0872 LDX #$00
0874 LDA $04FA,X
0877 JSR $FFD2
087A INX          'U1 2 0 18 2' zur Floppy schicken
087B CPX #$0B
087D BNE $0874
087F JSR $FFCC    Ausgabe wieder auf Bildschirm
0882 LDX #$01     und danach wieder auf Kanal
0884 JSR $FFC9    1 legen
0887 LDX #$00
0889 LDA $0505,X
088C JSR $FFD2
088F INX          'M-E'CHR$(3)CHR$(5) zur Floppy schicken
0890 CPX #$05
0892 BNE $0889
0894 JSR $FFCC    Ausgabe wieder normalisieren
0897 JSR $04AB    'M-R'CHR$(16)CHR$(0)CHR$(1) senden
                  (Kopierschutz,Knackmodul testen)
089A JSR $FFCC    Ausgabe normalisieren
089D LDX #$01     Ausgabe auf Kanal 1
089F JSR $FFC9    legen
08A2 LDA $0510    Disk initialisieren
08A5 JSR $FFD2
08A8 JMP $FFCC    Ausgabe normalisieren, Rücksprung

08AB LDX #$01     Ausgabe auf Kanal 1
08AD JSR $FFC9    legen
08B0 LDX #$00
```

```
08B2 LDA $050A,X
08B5 JSR $FFD2
08B8 INX          'M-R'CHR$(16)CHR(0)CHR(1)
08B9 CPX #$06
08BB BNE $08B2
08BD JSR $FFCC
08C0 LDX #$01     Eingabe auf Kanal 1
08C2 JSR $FFC6    stellen
08C5 JSR $FFCF    Zeichen von Floppy holen
08C8 CMP #$FF
                  auf Richtigkeit prüfen
08CA BNE $08CA    Endlosschleife, wenn Zeichen falsch
08CC RTS          sonst Rücksprung

08CD LDA #$01
08CF LDX #$08
08D1 LDY #$0F
08D3 JSR $FFBA
08D6 LDA #$00
08D8 JSR $FFBD
08DB JMP $FFC0  OPEN 1,8,15

08DE LDA #$02
08E0 LDX #$08
08E2 TAY
08E3 JSR $FFBA
08E6 LDA #$02
08E8 LDX #$F8
08EA LDY #$04
08EC JSR $FFBD
08EF JMP $FFC0  OPEN 2,8,2,"#2", Rücksprung

08F2 JSR $04CD   OPEN 1,8,15
08F5 JMP $04AB   Rückmeldung von Floppy abfragen

08F8 23 32 55 31 20 32 20 30 #2U1 2 0
0900 20 31 38 20 32 4D 2D 45  18 2M-E
0908 03 05 4D 2D 52 10 00 01   M-R
0910 49 0C 80 09 80 C3 C2 CD I    CBM
0918 38 30 4C 81 EA 02 00 00 80
```

Nachdem das Programm soweit geschrieben ist, daß es den Kopierschutz abfragt, sich entcodiert und sich selbst in den Bildschirm schreibt, müssen wir es entsprechend codieren.

Zu diesem Zweck benutzen wir, wie bereits gesagt, den Step-Codierer aus Kapitel 3.3.4, mit dem wir den Speicherbereich von $081B bis 0838 codieren. Der nächste Schritt ist das Codieren des restlichen Programms mit der folgenden Routine:

```
1008 LDX #$00     Zähler auf Null setzen
100A LDA $083B,X  Byte laden
100D EOR #$45     mit #$45 verknüpfen
100F STA $083B,X  und speichern
1012 DEC $083B,X  zusätzlich noch um Eins verringern
1015 INX          Zähler erhöhen
1016 BNE $100A    verzweige, wenn noch Bytes zu codieren
1018 RTS          Rücksprung
```

Danach hängen wir das Step-Entcodierprogramm an das Ende des zu schützenden Programms ab $091E an. Dieses Step-Programm wird beim Start nach $C000 kopiert und dort aufgerufen.

Nachdem wir jetzt unser Programm vollständig codiert haben, versehen wir es noch mit einem Autostart, wie er in Kapitel 3 vorgestellt wird. Zum guten Schluß müssen wir noch die Diskette mit dem abzufragenden Kopierschutz versehen. In diesem Fall haben wir den Kopierschutz aus Kapitel 6.6.3 gewählt und dessen Abfrage mit Hilfe des dort vorgestellten Programms auf Track 18 Sektor 2 gespeichert.

Das Hauptprogramm, das in diesem Beispiel geschützt werden soll, ist der Teil von $085E bis $0864, der lediglich die Rahmenfarbe erhöht und testet, ob ein Knackmodul zum Knacken eingesetzt wurde.

11. Was ein Software-Haus über "Cracker" wissen sollte

In diesem Kapitel wollen wir deutlich machen, wie Knacker bei ihrer Arbeit vorgehen, so daß Sie sich beim Entwurf Ihres Schutzes daran orientieren können. Bitte verstehen Sie folgende Erläuterung nicht als Anleitung zur Erstellung von Raubkopien, auch wenn wir es manchmal nicht vermeiden konnten, Informationen zu liefern, die einem Knacker dienlich sein könnten.

Es gibt für einen "Cracker" mehrere Möglichkeiten, an ein geschütztes Programm heranzugehen. Welche dieser Möglichkeiten er verwendet, hängt in erster Linie von seinem jeweiligen Wissensstand ab.

Der "Möchtegern-Knacker", der sich lediglich mit der Bedienung seines Kopierprogramms oder seines Knackmoduls auskennt, gehört zu der für uns ungefährlichsten Gruppe. Gegen ihn kann man sich schon durch einen einfachen Kopier- oder Knackmodulschutz sichern.

Der "Durchschnitts-Knacker" besitzt schon etwas Erfahrung im Umgang mit Maschinensprachemonitoren. Eine sehr beliebte Knackmethode ist es, mit Hilfe eines RESET-Schalters das Programm zu verlassen und dann die benutzten Teile des Speichers abzusichern. Nachträglich wird dann die Startadresse ermittelt. Diese Art des Knackens ist seit Erscheinen von Knack-Betriebssystemen wieder sehr in Mode gekommen. Um sich gegen diese Leute zu wehren, muß man schon etwas mehr Aufwand treiben.

Zum Ermitteln der Startadresse durchsucht der "Durchschnitts-Knacker" mit seinem Monitor den Speicher nach einem möglichen Programmanfang. Meistens testet er dabei die Vielfachen von $1000 (=4096) durch. Um es ihm nicht zu einfach zu machen, sollten Sie Ihr Programm nach Möglichkeit nicht bei einer dieser Adressen beginnen lassen. Wenn Ihr Programm eine Unterroutine enthält, die nur ganz am Anfang ein einziges Mal

aufgerufen wird, so sollten Sie diese Routine hinterher löschen, damit sich das Programm nach einem RESET nicht ein zweites Mal starten läßt.

Sie können allerdings auch anders vorgehen, um einem solchen Knacker "das Leben schwer zu machen". Das Einbringen einer "CBM80"-Erkennung reicht allerdings oft nicht aus, um einen RESET-Schalter funktionsuntüchtig zu machen, da Betriebssysteme, die diese Erkennung umgehen, inzwischen relativ weit verbreitet sind. Daher ist es sicherer, Daten oder Programmteile in Speicherbereichen unterzubringen, die bei einem RESET mit Sicherheit gelöscht werden (siehe Kapitel 7.1).

In den meisten Fällen ist dieser Schutz schon hinreichend. Viele Knacker nutzen jedoch gerade das "CBM80" dazu aus, beispielsweise den Bildschirmspeicher mit Hilfe eines RESETs abzusichern. Dazu lassen sie den RESET-Vektor der Erkennung auf eine eigene Routine zeigen, die ihnen die sonst verlorenen Daten in einen freien Teil des Speichers schiebt. Es ist dazu noch nicht einmal unbedingt erforderlich, diese Erkennung zu verwenden. Denselben Effekt erzielt man auch durch Ändern des NMI-Vektors $0318/$0319 (792/793) und Betätigen der RESTORE-Taste. Das funktioniert sogar dann, wenn das zu "knackende" Programm die "CBM80"-Erkennung benutzt, da der Vektor $0318/$0319 höhere Priorität gegenüber dem NMI-Vektor der Erkennung besitzt. Hiergegen schützt man sich durch Auffüllen aller vom Programm nicht benötigten Speicherbereiche, wodurch der Knacker keinen Platz für seine Verschiebe-Routine hat.

Weiterhin sollten Sie beim Schutz Ihrer Programme daran denken, daß ein Knacker mit einem Diskmonitor das File auf der Diskette nach der "CBM80"-Erkennung oder nach anderen markanten Bytes durchsuchen und diese ändern kann. Der einfachste Schutz dagegen ist eine Codierung Ihrer Programmdaten oder eine Prüfsumme (siehe Kapitel 3.2/3.3), anhand derer Sie die Änderungen erkennen und abfangen können.

Die gezielte Suche nach Schutzroutinen ist sowieso eine beliebte Methode, um sich die Arbeit des exakten Nachverfolgens eines Programmschutzes zu ersparen. Dieses wird den Knackern be-

sonders dadurch erleichtert, daß nahezu alle Maschinensprachemonitore über Suchbefehle verfügen. Eine Decodierroutine finden sie beispielsweise, indem sie nach dem einzigen uncodierten Programmteil Ausschau halten. Wenn man allerdings seine Decodierroutine mit illegalen Opcodes (siehe 3.4) ausstattet, so kann ein Knacker nicht mehr ohne weiteres die codierten von den uncodierten Programmteilen unterscheiden.

Die Grundlage zur Durchsuchung eines Programmteils ist, diesen Programmteil zuerst einmal auf der Diskette zu finden. Der Directory-Schutz hat die Aufgabe, gerade das zu verhindern. Jegliche Manipulation am Inhaltsverzeichnis der Diskette läßt sich jedoch mit einem Diskmonitor oder einem Directory-Editor wieder entfernen. Wir müssen nämlich davon ausgehen, daß unter Knackern solche Programme weit verbreitet sind. Alle uns bekannten Tricks beruhen schließlich auf Änderungen am Filenamen, am Diskettennamen oder an den Blockverbindungszeigern, was bedeutet, daß kein Schutz gegenüber einem Diskmonitor wirksam bleibt. Da jeder den exakten Aufbau des Directorys schon aus dem mitgelieferten Handbuch zur 1541 entnehmen kann, ist dieser auch relativ bekannt.

Die einzige Methode, ein File so zu verstecken, daß man es nicht wiederfindet, ist, es mit Direktzugriffsbefehlen frei über die Diskette zu verteilen, wodurch es sich nur über ein eigenes Ladeprogramm wieder einladen läßt. Eine Erweiterung dieses Schutzes ist die Verwendung eines eigenen Diskettenformates. So wird der Knacker dazu gezwungen, sich mit der Laderoutine auseinanderzusetzen.

Findigen Knackern gelingt es auch, ein nicht laufendes Programm, das von einem "Original" kopiert wurde, trotzdem zum Laufen zu bringen, indem sie einfach die Kopierschutzabfrage von der kopierten Diskette entfernen. Eine Schutzmethode dagegen ist, Daten vom Kopierschutz zu holen und diese später im Programm noch einmal zu verwenden.

Überhaupt ist es für einen Knacker relativ schwer, Daten, Werte oder Vektoren, die im Zusammenhang mit dem Kopierschutz geholt, gesetzt oder berechnet werden (zum Beispiel als Prüf-

summe) und die für das geschützte Programm "lebenswichtig" sind, herauszufinden und dementsprechend diese Werte zu rekonstruieren. Wer sich hier nicht an die genaue Analyse des Programmschutzes herantraut, hat kaum eine Chance, das Programm zu knacken.

Entsprechend sind Kopierschutzabfragen, die mitten im Programm benutzt werden, wesentlich schwerer zu finden als solche, die direkt am Programmanfang aufgerufen werden. Man sollte sich aber nicht zu sicher sein, daß so eine Abfrage nicht doch gefunden wird, wenn man sie nicht gesondert schützt. Beispielsweise kann der Knacker eine Dongle-Abfrage (Kapitel 3.5) sehr leicht ausschalten, indem er das Programm nach allen Befehlen durchsucht, die die entsprechenden Ein-/Ausgabe-Ports adressieren. Besonders leicht wird es für einen Knacker, dessen Knackmodul zwar an einem Programm gescheitert ist, das mit einer der in Kapitel 7 beschriebenen Methoden gesichert ist, wenn diese Schutzabfrage zu leicht "von Hand" entfernbar ist. Genauso wie bei der Dongle-Abfrage sucht er ebenfalls nach auf die Register des SIDs oder der Echtzeituhren zurückgreifenden Befehlen. Sie sollten daher solche Abfragen codiert im Programm stehen lassen und immer nur kurzzeitig vor dem eigentlichen Aufruf decodieren.

Die gefährlichste Gruppe von Knackern bilden die "eingefleischten Freaks", die nicht davor zurückschrecken, sich stunden- und tagelang mit einem Bleistift und Papier bewaffnet an den Rechner zu setzen und den Programmschutz bis ins kleinste Detail zu sezieren. Um sich auch hiergegen zu wehren, reicht zum Beispiel eine einfache Codierung bei weitem nicht aus.

Ein Autostart läßt diese Leute weitgehend kalt, da man ihn grundsätzlich mit denselben Methoden entfernen kann, mit denen man ihn aufgetragen hat. Beispielsweise erzeugen wir ja einen Autostart auf Diskette durch nachträgliches Ändern der Startadresse des Programms mit der Routine aus Kapitel 3.1.4. Umgekehrt geht nun ein Knacker hin und stellt genauso die Startadresse um, allerdings auf einen freien Bereich. Ent-

sprechend verfährt der Knacker bei einem Autostart von Kassette (siehe Kapitel 5.3). Prinzipiell können wir uns nicht dagegen schützen.

Die nächste Hürde, die allerdings nur dann auftaucht, wenn wir ein BASIC-Programm geschützt haben, ist der List-Schutz. Dummerweise ist die Anzahl der möglichen List-Schutzsysteme relativ gering und beschränkt sich nahezu auf die in diesem Buch beschriebenen Verfahren. Ein erfahrener Knacker kennt selbstverständlich diese Methoden. Es existieren sogar einige Betriebssysteme, die viele List-Schutzverfahren außer Kraft setzen. Wer sich zudem noch mit dem Aufbau eines BASIC-Programms im Speicher auskennt, wie er in Systemhandbüchern wie dem "64 Intern" beschrieben ist, kann mit Hilfe eines Monitors auch alle restlichen Schutzsysteme überwinden. Daher ist auch ein List-Schutz noch nicht optimal.

Es gibt allerdings einige wirksame Methoden, dem Knacker das Nachverfolgen zu erschweren. Eine besteht darin, illegale Opcodes zu verwenden. Wer diese Codes nicht kennt, hat keine Möglichkeit, sich das Programm anzusehen. Aber auch für den, der weiß, welche Bedeutung diese Codes haben, stellen sie ein Handikap bei der Analyse dar. Allerdings ist es natürlich genauso schwer, solchen Code zu schreiben, wie ihn hinterher wieder zu entziffern. Daher muß man schon ziemlichen Aufwand mit den "Illegals" treiben, bevor ein "echter" Knacker aufgibt. Gleiches gilt für alle anderen Verfahren, bei denen man versucht, durch einen möglichst komplizierten und verschachtelten Programmablauf, der außerdem noch von mehrfachen Codierungen und Sicherheitsabfragen begleitet wird, den Knacker zu verwirren. Im Zweifelsfalle findet sich nämlich doch irgendwann einmal ein Knacker, der eine größere Ausdauer als der Konstrukteur des Schutzes besitzt.

Die Knacker haben in Bezug auf das Nachverfolgen von Programmen noch einen anderen Vorteil als den, daß sie sich gegenüber den Schützern in der Überzahl befinden. Für den C64 existieren nämlich einige sogenannte "Einzelschritt-Simulatoren", mit deren Hilfe man jeden einzelnen Schritt eines Maschinenprogrammes beobachten kann. Diese Programme sind aber rela-

tiv leicht auszutricksen. Die sicherste Methode dagegen ist, alle unbenutzten Speicherbereiche zu überschreiben oder die Interruptvektoren $0314/$0315 beziehungsweise $0318/$0319 zu initialisieren.

Die einzige uns bekannte Möglichkeit, mit relativ geringem Aufwand einen extrem schwer zu entschlüsselnden Code zu produzieren, stellt das Compilieren eines BASIC-Programms dar. Der von einem Compiler erzeugte P-Code ist dermaßen schwer zu verstehen, daß selbst hartgesottene Knacker daran verzweifeln. Die meisten Knacker würden bei einem compilierten Programm zuerst einmal versuchen, mit einem Diskmonitor oder Monitor eventuelle Floppy-Befehle im P-Code zu finden. Die Arbeit, Texte wie zum Beispiel "B-E" zu verschlüsseln, nimmt uns nämlich der Compiler nicht ab. Floppy-Programme sollte man daher nicht ungeschützt auf einem Block der Diskette unterbringen, sondern am besten als codierte Daten mit in das compilierte BASIC-Programm übernehmen.

Es ist natürlich nicht auszuschließen, daß ein sehr versierter Knacker den P-Code des verwendeten Compilers analysiert. Dies sind und bleiben jedoch Einzelfälle, die normalerweise nicht zu einer ausgedehnten Verbreitung des Programms als Raubkopie führen. Gefährlich wird es nur, wenn ein solcher Knacker sein Wissen dazu nutzt, einen Recompiler zu schreiben, mit dessen Hilfe jedermann ein compiliertes Programm wieder in ein BASIC-Programm umwandeln kann. Solche Fälle sind tatsächlich schon vorgekommen. Achten Sie also bei der Wahl ihres Compilers darauf, daß dazu noch kein solcher Recompiler existiert.

12. Kopierschutz auf dem C128

Alle in diesem Buch beschriebenen Kopierschutzverfahren lassen sich selbstverständlich auch auf dem C128, dem Nachfolgemodell des C64, verwenden, sofern man ihn in den 64er-Modus schaltet und statt einer 1570/1571-Diskettenstation eine 1541 anschließt. Uns interessiert aber vielmehr, wie man Kopierschutz im 128er-Modus und mit einer 1570 beziehungsweise 1571 betreibt.

12.1 Übereinstimmungen und Unterschiede zum C64

Da der C128 der Nachfolger des C64 ist, stimmt er in vielen Strukturen mit ihm überein. Daher funktionieren viele Programmschutzsysteme ohne oder mit wenigen Änderungen auch auf dem neuen Rechner. Unterschiede treten bei der Verwaltung des BASIC-Speichers, der RAM-Bänke und in einigen Fällen auch bei Betriebssystemroutinen auf. Außerdem ist die Floppy 1571 nicht völlig kompatibel zur 1541, was auch hier Änderungen notwendig werden läßt. Im folgenden werden wir erst einmal beschreiben, wie man die meisten der bisher abgedruckten Programme auch auf dem C128 zum Laufen bringt.

Der C128 ist ein Rechner, der wesentlich mehr Speicherplatz verwalten muß als der C64. Daher stellt das BASIC uns den Befehl "BANK" zur Verfügung, mit dessen Hilfe wir durch die Befehle "SYS", "PEEK" und "POKE" den gesamten Speicher erreichen können. Programme aus diesem Buch, die Werte in den VIC, den SID oder die CIAs "poken", funktionieren ohne Änderung, da das BASIC als voreingestellten Wert für den "BANK"-Befehl den Wert 15 vorgibt und diese Chips dann im gleichen Bereich wie im 64er liegen. Sollten Sie allerdings in Ihren Programmen Bank-Umschaltungen vornehmen, so denken Sie bitte daran, durch "BANK 15" vor entsprechenden "PEEK"- und "POKE"-Sequenzen wieder den richtigen Bereich anzuwählen. Auf Einzelheiten werden wir später noch zu sprechen kommen.

Ein weiteres Problem betrifft die Unterbringung von Maschinenprogrammen. Der Bereich $C000 (=49152) bis $CFFF (=53247) steht uns leider nicht mehr in der Weise zur Verfügung wie im C64. Dagegen finden wir in der RAM-Bank 0 unterhalb des BASIC-Speichers jede Menge freien Speicher. Folgende Bereiche stehen uns zur Verfügung:

```
Hex
Dez
$0B00 - $0BFF  2816 - 3071  Kassettenpuffer
$0C00 - $0DFF  3072 - 3327  RS232-Puffer
$0E00 - $0FFF  3584 - 4095  Bereich für Sprite-Daten
$1300 - $17FF  4864 - 6143  unbenutzter Bereich (!)
$1800 - $1BFF  6144 - 7167  RAM für Funktionstasten-
                            anwendungen, normalerweise frei
```

Zur Kompatibilität der Floppy 1571 zur 1541 ist zu sagen, daß sich die meisten Schwierigkeiten dadurch umgehen lassen, indem man die Floppy in den 1541-Modus schaltet.

Zu Kapitel 2:

Die meisten Listschutzsysteme funktionieren nicht nur auf dem C64, sondern auch auf dem 128er. Im folgenden haben wir die wichtigsten Unterschiede des alten Rechers zum neuen aufgelistet.

Zu Kapitel 2.1.1:

Der Listschutz mit SHIFT-L klappt leider nicht mehr, da der BASIC-Interpreter des C128 etwas anders aufgebaut ist als der des C64.

Zu Kapitel 2.1.2:

Diese Listschutzmethode funktioniert unverändert auch auf dem 128er.

Zu Kapitel 2.1.3:

Die Adresse des LIST-Vektors hat sich beim C128 nicht verändert, jedoch die möglichen Adressen, auf die er zeigen kann. Um den LIST-Vektor auf die RESET-Routine zeigen zu lassen, benötigt man folgende POKEs:

```
POKE 774,61:POKE 775,255
```

Zu Kapitel 2.1.4:

Beim LIST-Schutz "Nur Zeilennummern" hat sich prinzipiell nichts verändert. Nur der BASIC-Start liegt jetzt nicht mehr bei $0800, sondern bei $1C00. Außerdem besitzt der 128er schon einen eingebauten Monitor, mit dem man die nötigen Änderungen vornehmen kann.

Zu Kapitel 2.1.5/2.1.6

Hier gilt das gleiche, was schon zu Kapitel 2.1.4 angemerkt wurde.

Zu Kapitel 2.1.7:

Diese Schutzmethode hat sich im Bezug zum C64 nicht verändert, nur das Schutzprogramm muß dem 128er angepaßt werden.

Hier das Schutzprogramm für den C128:

Hauptprogramm

```
1300 LDX #$00       Zähler auf Null
1302 LDA $1380,X    BASIC-Zeile
1305 STA $1C01,X    ins
```

```
1308 INX           RAM kopieren
1309 CPX #$28      28 Bytes erreicht?
130B BNE $1302     verzweige, wenn nicht erreicht
130D LDA #$93      Bildschirm
130F JSR $FFD2     löschen
1312 LDX #$00      PROGRAMMNAME:
1314 LDA $13A8,X   auf
1317 JSR $FFD2     Bildschim ausgeben
131A INX           Zähler erhöhen
131B CPX #$0E      schon 15 Bytes?
131D BNE $1314     verzweige, wenn nicht erreicht
131F LDX #$00      Zähler auf Null
1321 JSR $FFCF     Zeichen von Tastatur holen
1324 STA $13B8,X   und zwischenspeichern
1327 INX           Zähler erhöhen
1328 CMP #$0D      mit RETURN vergleichen
132A BNE $1321     verzweige, wenn nicht gedrückt
132C DEX           Return-Code abschneiden
132D STX $72       Zähler speichern
132F LDX #$08      Geräteadresse
1331 LDY #$00      und Sekundäradresse laden
1333 JSR $FFBA     File-Parameter übergeben
1336 LDA $72       Länge des Filenamens laden
1338 LDX #$B8      Adresse des Filenamens
133A LDY #$13      in LOW-HIGH-Byte Format laden
133C JSR $FFBD     Filenamen-Parameter übergeben
133F LDA #$00      Flag für LOAD
1341 LDX #$29      Startadresse
1343 LDY #$1C      bestimmen
1345 JSR $FFD5     LOAD
1348 LDX $72       Länge des Filenamen laden
134A LDA #$2F      Code für '/' laden
134C STA $13B8,X   und speichern
134F LDA #$53      Code für 'S' laden
1351 STA $13B9,X   und speichern
1354 INX           Zähler erhöhen
1355 INX           Zähler erhöhen
1356 STX $72       und speichern
1358 LDX #$08      Geräteadresse laden
135A LDY #$01      Sekundäradresse laden
```

```
135C JSR $FFBA      File-Parameter übergeben
135F LDA $72        Länge des Filenamens laden
1361 LDX #$B8       Adresse des Filenamens im
1363 LDY #$13       LOW-HIGH-Byte Format laden
1365 JSR $FFBD      Filenamen-Parameter übergeben
1368 LDX #$01       Startadresse im LOW-HIGH
136A LDY #$1C       Format laden
136C STX $FB        und
136E STY $FC        Speichern
1370 LDA #$FB       Zeiger auf Startadresse laden
1372 LDX $AE        Endadresse im
1374 LDY $AF        LOW-HIGH-Byte Format bestimmen
1376 JSR $FFD8      SAVE
1379 JMP $4F4F      BASIC-Zeilen neu binden und Rücksprung
```

BASIC-Zeile:

```
137C 00 00 00 00 0C 0C 0A 00 ........
1384 9E 20 37 31 38 34 00 00 . 7184..
138C 00 00 00 18 A9 28 65 2D ....)(e-
```

Programm hinter Sys-Zeile:

```
1390 LDA #$28       Länge des Programms laden
1392 ADC $2D        zu LOW-Byte addieren
1394 STA $2D        und speichern
1396 LDA #$00       falls
1398 ADC $2E        Übertrag
139A STA $2E        HIGH-Byte +1
139C JSR $4F4F      BASIC-Zeilen neu binden
139F JSR $AF7E      CLR
13A2 JMP $4AFD      RUN

13A5 00 00 00 50 52 4F 47 52 ...PROGR
13AD 41 4D 4D 4E 41 4D 45 3A AMMNAME:
13B5 20 00 00
```

Nachfolgend der BASIC-Loader:

```
100 FORI=1TO192STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE4863+I+J,A:NEXT:READA:
IFC=ATHENC=0:NEXT:END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATAA2,00,BD,80,13,9D,01,1C,E8,E0,28,D0,F5,A9,93, 157
301 DATA20,D2,FF,A2,00,BD,A8,13,20,D2,FF,E8,E0,0E,D0, 162
302 DATAF5,A2,00,20,CF,FF,9D,B8,13,E8,C9,0D,D0,F5,CA, 58
303 DATA86,72,A2,08,A0,00,20,BA,FF,A5,72,A2,B8,A0,13, 63
304 DATA20,BD,FF,A9,00,A2,29,A0,1C,20,D5,FF,A6,72,A9, 193
305 DATA2F,9D,B8,13,A9,53,9D,B9,13,E8,E8,86,72,A2,08, 110
306 DATAA0,01,20,BA,FF,A5,72,A2,B8,A0,13,20,BD,FF,A2, 28
307 DATA01,A0,1C,86,FB,84,FC,A9,FB,A6,AE,A4,AF,20,D8, 1
308 DATAFF,4C,4F,4F,00,00,00,00,0C,0C,0A,00,9E,20,37, 0
309 DATA31,38,34,00,00,00,00,00,18,A9,28,65,2D,85,2D, 202
310 DATAA9,00,65,2E,85,2E,20,4F,4F,20,7E,AF,4C,FD,4A, 141
311 DATA00,00,00,50,52,4F,47,52,41,4D,4D,4E,41,4D,45, 134
312 DATA3A,20,00,00,50,52,4F,42,45,2F,53,00,CA,3D,CA, 37
```

Noch ein wichtiger Hinweis:

Nach dem Einlesen der DATA-Zeilen in den Speicher muß unbedingt ein RESET durchgeführt werden, da sonst die Funktionsweise des Schutzprogramms beeinträchtigt wird.

Das zu schützende Programm muß sich vorher abgespeichert auf Diskette befinden. Das Schutzprogramm startet sich anschließend mit SYS DEC("1300"). Nach dem Start fragt das Programm nach dem Filenamen des zu schützenden BASIC-Programms. Sobald das Schutzprogramm das BASIC-Programm eingeladen hat, wird es sofort wieder mit dem gleichen Filenamen auf Diskette gespeichert. Nur wird hinter dem Filenamen noch ein /S angehängt. Deshalb sollte der Filename nicht länger als 14 Zeichen sein.

Zu Kapitel 2.2.1:

Bei dieser Schutzmethode hat sich außer den veränderten Zeigeradressen, die jetzt auf $1210 $1211 liegen, nichts verändert.

Zu Kapitel 2.2.2:

Hier gilt das gleiche, was schon zu Kapitel 2.1.7 angemerkt wurde.

Nachfolgend das Schutzprogramm für den C128:

```
1300 LDX #$00       Zähler auf Null
1302 LDA $1380,X    BASIC-Zeile
1305 STA $1C01,X    ins RAM kopieren
1308 INX            Zähler erhöhen
1309 CPX #$3B       schon alle Bytes kopiert?
130B BNE $1302      verzweige, wenn nicht
130D JSR $C142      Bildschirm löschen
1310 LDX #$00       PROGRAMMNAME:
1312 LDA $13C0,X    auf dem
1315 JSR $FFD2      Bildschirm bringen
1318 INX            Zähler erhöhen
1319 CPX #$0E       schon 15 Bytes?
131B BNE $1312      verzweige, wenn nicht
131D LDX #$00       Zähler auf Null
131F JSR $FFCF      Zeichen von Tastatur holen
1322 STA $13D0,X    und zwischenspeichern
1325 INX            Zähler erhöhen
1326 CMP #$0D       Vergleich mit RETURN
1328 BNE $131F      verzweige, wenn kein RETURN-Code
132A DEX            RETURN-Code abschneiden
132B STX $72        und Zähler speichern
132D LDX #$08       Geräteadresse laden
132F LDY #$00       Sekundäradresse laden
1331 JSR $FFBA      Fileparameter übergeben
1334 LDA $72        Länge des Filenamens laden
1336 LDX #$D0       Adresse des Filenamens im
```

```
1338 LDY #$13       LOW-HIGH-Byte Format laden
133A JSR $FFBD      Filenamen-Parameter übergeben
133D LDA #$00       Flag für LOAD
133F LDX #$3C       Startadresse
1341 LDY #$1C       bestimmen
1343 JSR $FFD5      LOAD
1346 NOP
1347 NOP
1348 NOP
1349 NOP
134A NOP
134B NOP
134C NOP
134D NOP
134E LDX $72        Länge des Filenamens laden
1350 LDA #$2F       Code für '/' laden
1352 STA $13D0,X    und speichern
1355 INX            Zähler erhöhen
1356 LDA #$41       Code für 'A' laden
1358 STA $13D0,X    und speichern
135B INX            Zähler erhöhen
135C STX $72        und speichern
135E LDX #$08       Geräteadresse laden
1360 LDY #$01       Sekundäradresse laden
1362 JSR $FFBA      File-Parameter übergeben
1365 LDA $72        Länge des Filenamens laden
1367 LDX #$D0       LOW-HIGH-Byte Adressformat laden
1369 LDY #$13       und
136B JSR $FFBD      übergeben
136E LDX #$01       Startadresse
1370 LDY #$1C       laden
1372 STX $FB        und
1374 STY $FC        speichern
1376 LDA #$FB       Zeiger auf Startadresse laden
1378 LDX $AE        Endadresse
137A LDY $AF        bestimmen
137C JSR $FFD8      SAVE
137F JMP $4F4F      BASIC-Zeilen neu Bilden und Rücksprung
```

BASIC-Zeile und Bildschimtext:

```
1382 FF FF 8F 20 22 A2 01 8E   . ..
138A 03 1C CA 8E 04 1C A9 1C ..J...).
1392 8D 02 03 A9 1C 8D 03 03 ...)....
139A 60 A2 FF 8E 03 1C 8E 04 ' .....
13A2 1C A9 C6 8D 02 03 A9 4D .)F...)M
13AA 8D 03 03 4C C6 4D 00 3C ...LFM.<
13B2 08 02 00 9E 37 31 37 36 ....7176
13BA 00 00 00 00 00 00 50 52 ......PR
13C2 4F 47 52 41 4D 4D 4E 41 OGRAMMNA
13CA 4D 45 3A 20 00 00 51 D0 ME: ..QP
```

Nachfolgend der BASIC-Loader:

```
100 FORI=1TO208STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE4863+I+J,A:NEXT:READA: IFC=ATHENC=0:NEXT:
END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATAA2,00,BD,80,13,9D,01,1C,E8,E0,3B,D0,F5,20,42, 214
301 DATAC1,A2,00,BD,C0,13,20,D2,FF,E8,E0,0E,D0,F5,A2, 33
302 DATA00,20,CF,FF,9D,D0,13,E8,C9,0D,D0,F5,CA,86,72, 179
303 DATAA2,08,A0,00,20,BA,FF,A5,72,A2,D0,A0,13,20,BD, 60
304 DATAFF,A9,00,A2,3C,A0,1C,20,D5,FF,EA,EA,EA,EA,EA, 200
305 DATAEA,EA,EA,A6,72,A9,2F,9D,D0,13,E8,A9,41,9D,D0, 109
306 DATA13,E8,86,72,A2,08,A0,01,20,BA,FF,A5,72,A2,D0, 160
307 DATAA0,13,20,BD,FF,A2,01,A0,1C,86,FB,84,FC,A9,FB, 147
308 DATAA6,AE,A4,AF,20,D8,FF,4C,4F,4F,FF,FF,8F,20,22, 87
309 DATAA2,01,8E,03,1C,CA,8E,04,1C,A9,1C,8D,02,03,A9, 200
310 DATA1C,8D,03,03,60,A2,FF,8E,03,1C,8E,04,1C,A9,C6, 122
311 DATA8D,02,03,A9,4D,8D,03,03,4C,C6,4D,00,3C,08,02, 192
312 DATA00,9E,37,31,37,36,00,00,00,00,00,00,50,52,4F, 100
313 DATA47,52,41,4D,4D,4E,41,4D,45,3A,20,00,00,51,D0, 16
```

Zu Kapitel 2.3:

Um beim 128er die Stop-Taste außer Betrieb zu setzen, muß folgender POKE eingegeben werden:

```
POKE 808,107
```

Der Sprungvektor ist liegt wie beim C64 in der gleichen Adresse, nur der Zeiger des LOW-Bytes muß geändert werden.

Leider kann man diesen Schutz umgehen, wenn man beim Drücken des RESET-Knopfs gleichzeitig die Stop-Taste gedrückt hält.

Man hat anschließend das unversehrte BASIC-Programm, wenn der Benutzer den Monitor mit X verlassen hat. Deshalb ist es ratsam, zusätzlich noch einen RESET-Schutz einzubauen:

```
BANK 1
POKE 65528,33
```

Nach diesen beiden Befehlen hilft nur das Ausschalten des Rechners.

Zu Kapitel 2.4 und 2.4.1:

Dieses Verfahren funktioniert unverändert auch auf dem C128.

Zu Kapitel 3.1: Autostartmethoden

Allgemeine Anmerkungen

Prinzipiell unterscheiden sich die Methoden zur Erzeugung eines Autostarts beim C64 nicht wesentlich von denen des C128. Alle Ideen zum Autostart, die wir in Kapitel 3 beschrieben haben, lassen sich auch den C128 übertragen, wenn man einige Punkte berücksichtigt.

Als erstes ist zu beachten, daß die Speicherkonfiguration sich in wesentlichen Punkten ändert. Unter anderem ändert sich die Länge und die Adresse des Kassettenpuffers. Auch das ROM belegt beim C128 nicht genau denselben Bereich wie beim C64. Aus diesen Veränderungen folgt, daß sich auch die Adressen der ROM-Routinen und der Zeiger in der Zeropage ändern. Allgemein ist es empfehlenswert, die Zeiger ab $0300 und die Sprungtabelle am Ende des ROMs zu benutzen, da diese im wesentlichen identisch sind.

Ein Autostart auf dem C128, der nach einer in Kapitel 3 beschriebenen Methode arbeitet, ist nur als Schutzmethode sinnvoll, da der C128 über eine Boot-Routine verfügt, die wir unten erläutern werden. Diese Routine stellt eine sehr komfortable und einfache Möglichkeit dar, um einen Autostart zu erzeugen. Größter Nachteil dabei ist, daß diese Autostartprogramme leicht verstanden und modifiziert werden können. Um die Programme zu schützen, muß man auch hier auf die "alten" Methode zurückgreifen.

Boot-Sektor und -Routine

Im Gegensatz zum C64 verfügt der C128 bereits über eine Betriebssystemroutine, die zum automatischen Laden und Starten eines Programms dient, das auf Diskette vorliegt. Diese Routine wird sofort nach dem Einschalten des Rechners angesprungen, was sich durch ein kurzes Anlaufen der Floppy äußert. Erst danach meldet sich der Rechner mit dem blinkenden Cursor zur Stelle, um Befehle des Benutzers zu erwarten und seine eigentliche Arbeit aufzunehmen. Es ist dabei unerheblich, ob Sie bereits eine Diskette in das Laufwerk eingelegt haben.

Diese Routine sucht auf der Diskette nach einem sogenannten Boot-Sektor, in dem diverse Informationen über das zu ladende Programm oder die zu ladenden Blöcke stehen. Da die Boot-Routine natürlich nicht die ganze Diskette nach einem solchen Boot-Sektor absuchen kann, gibt es nur eine fest definierte

Stelle auf der Diskette, die als Bootsektor genutzt werden kann. Hierbei handelt es sich um:

Seite 0, Spur 1, Sektor 1

Vorsicht ist geboten, wenn Sie einen eigenen Boot-Sektor auf einer Diskette installieren wollen. Auch wenn der Boot-Sektor physikalisch der erste Block auf der Diskette ist, kann es doch vorkommen, daß dieser Platz schon von einem anderen Programm belegt ist. Vor dem Installieren sollten Sie immer erst prüfen, ob der Block nicht schon benutzt wurde.

Um den Aufbau des Bootsektors verstehen zu können, sollten Sie den schematischen Ablauf der im Kernal verankerten Routine kennen, der folgendermaßen abläuft:

1. Im DOS-Puffer der erweiterten Zeropage wird ein Block-Read-Befehl auf die Spur 1, Sektor 1 aufgebaut.

2. Der Befehl wird ausgeführt und der gelesene Block, sofern sich eine formatierte Diskette im Laufwerk befindet, in den Kassettenpuffer geladen.

3. Anhand der ersten drei Bytes des gelesenen Blocks wird geprüft, ob es sich um einen Boot-Sektor handelt. Der Identifizierungscode lautet CBM, die ersten drei Bytes müssen also die Werte $43, $42 und $4D haben. Ist dieser Code nicht vorhanden, wird die Boot-Routine abgebrochen.

4. Die auf den CBM-Code folgenden vier Bytes werden in vier Zeropage-Adressen geladen. Normalerweise sind diese vier Bytes auf den Wert $00 gesetzt. Die ersten beiden Bytes können eine Startadresse enthalten, die jedoch nichts mit der Anfangsadresse zu tun hat, an die das gewünschte Programm geladen werden soll. Das dritte Byte ist der entsprechende Konfigurationsindex zur genannten Startadresse. Das nun folgende vierte Byte gibt die Anzahl der Blöcke an, die außer dem Boot-Sektor noch von Diskette geladen werden sollen. Weist dieses Byte den Wert

$00 auf, so sind die oben erwähnten Bytes für die Startadresse und das dazugehörige Konfigurations-Byte unerheblich, da kein Programm nachgeladen wird.

5. Unabhängig von der Anzahl der zu ladenden Blöcke, werden die nun folgenden Bytes, also ab dem 8. Byte des Blocks, über die BSOUT-Routine auf dem Bildschirm ausgegeben. Hiermit läßt sich der Bildschirm löschen oder ein beliebiger Text auf dem Bildschirm ausgeben. Die Zeichenausgabe erfolgt so lange, bis die Endmarkierung $00 gefunden wird.

6. Nun bekommen die unter Punkt 4 erwähnten Steuer-Bytes eine Bedeutung, sofern der Blockzähler nicht auf den Wert $00 gesetzt ist. In diesem speziellen Fall wird die nun folgende Routine übersprungen. Ist dies aber nicht der Fall, wird im DOS-Befehlspuffer einer neuer Befehls-String gebildet, der weitere Blöcke nachlädt. Die Bestimmung der Folgeblocks fällt dabei denkbar einfach aus: Die Sektornummer wird immer um eins erhöht, es wird also der nächste Block der Spur eingelesen. Da nur 21 Sektoren, mit den Nummern 0 bis 20, existieren, wird bei der Sektornummer 21 die Spur um eins erhöht. Danach werden die weiteren Blöcke von der neuen Spur wieder ab Sektor 0 eingelesen. Wie schon oben erwähnt, ist die Anzahl der zu lesenden Blöcke im ersten Boot-Block festgelegt. Die Speicheradresse, an die die Blöcke geladen werden, wird durch die ersten drei Bytes des zuerst nachgeladenen Blocks bestimmt. Die ersten beiden Bytes geben dabei die Speicheradresse an. In dem dritten Byte ist die Konfiguration gespeichert. Alle weiteren Blöcke werden im Speicher hinter dem vorherigen abgelegt.

7. Anschließend setzt die Boot-Routine wieder hinter die (wenn vorhanden) Textkonstanten des ursprünglichen Boot-Blocks im Kassettenpuffer an. Hier kann nun ein Dateiname stehen, wie er auch im Directory der Diskette vorhanden ist. Mit Ausnahme der Tatsache, daß die den Dateinamen bildenden Zeichen nicht auf dem Bildschirm ausgegeben werden, werden auch hier alle Bytes gelesen,

bis die Boot-Routine abermals auf das Endezeichen $00 stößt. Die Länge des Dateinamens wird dabei in einem Zeiger festgehalten.

8. Ist die Länge des Dateinames im Speicher größer null, so werden dem Namen die Zeichen "0:" vorangestellt und die Länge des Namens entsprechend erhöht. Danach wird das entsprechende Programm in den Speicher eingeladen. Wenn das geschehen ist, oder wenn die Länge des Dateinamens mit null festgehalten wurde, so setzt die Boot-Routine wieder hinter dem $00 Trenncode ein, der das Ende des Dateinamens kennzeichnet.

9. Die auf den Dateinamen folgenden Bytes werden nun als Maschinenprogramm des Boot-Sektors angesehen, und die Bootroutine übergibt die Steuerung an dieses Programm. Von nun an können Sie den gesamten weiteren Ablauf selbst bestimmen. Es steht Ihnen dabei offen, ob Sie weitere Programme nachladen oder eventuell eingeladene Blöcke oder Programme an einer bestimmten Stelle starten.

Wenn Sie bei der Erstellung des Bootsektors die oben genannten Schritte des Betriebssystems beachten, so werden Sie bald feststellen, daß es gar nicht so schwer ist, eigene Bootsektoren zu erzeugen und diese auch sinnvoll einzusetzen.

Hier noch einmal die wichtigsten Merkmale und Anordnungen:

```
Byte 0, 1, 2:   CBM-Identifikationscode
Byte 3, 4:      Startadresse für Folge-Boot-Sektoren
Byte 5:         Konfigurationsindex
Byte 6:         Blockzähler für die Anzahl der Folgesektoren
Byte 7:         Bis zum ersten Trenncode $00 eigener Text
Dann:           Name des nachzuladenden Programms mit $00
Dann:           Eigenes Maschinenprogramm
```

Zu Kapitel 3.2: Prüfsummen und Selbstzerstörung

Die in diesem Kapitel erklärten Schutzsysteme sind nicht rechnerspezifisch. Daher braucht für den C128 nichts geändert werden.

Zu Kapitel 3.3: Codierung

Allgemein können die Codiersysteme aus Kapitel 3, die für den C64 entwickelt wurden, auch auf den C128 übernommen werden. Da diese Programme nicht mit den Betriebssystemroutinen arbeiten, treten bei der Verwendung des anderen Betriebssystems keine Probleme auf.

Auch die Timercodierung kann unverändert übernommen werden, da es sich beim C128 um die gleichen Register handelt.

Zu Kapitel 3.4: Illegal-Opcodes

Der Prozessor des C128 unterscheidet sich so wenig vom Prozessor des C64, daß er sich auch bei illegalen Opcodes genauso verhält. Daher ergeben sich hier keine Änderungen.

Zu Kapitel 3.5: Dongle

Wenn man beachtet, daß man vor Abfrage der Joystick-Ports mit dem Befehl "BANK 15" die richtige Speicherbank anwählt, braucht man an den bestehenden Programmen nichts zu ändern.

Zu Kapitel 3.6:

Die in diesem Buch aufgeführten Schutzmethoden funktionieren auch auf dem 128er. Die dazugehörigen Beispielprogramme müssen nur den Speicheradressen des C128 angepaßt werden.

Hier nun die modifizierten Beispielprogramme:

Pass 1

```
10 POKE 808,98:PRINT CHR$(147):A=0
20 POKE 19,1
30 PRINT CHR$(19):INPUT "PASSWORD :";A$
40 POKE 19,0:PRINTCHR$(13)
50 IF A$="" THEN 20
60 IF A$="HARALD" THEN PRINT CHR$(17);"PASWORD AKZEPTIERT!":END
70 A=A+1:IF A=3 THEN SYS 65341
80 GOTO 20
```

Pass 2

```
1300 JSR $C142      Bildschirm löschen
1303 LDX #$00       Zähler auf Null
1305 LDA $1363,X    PASSWORD: auf Bildschim
1308 JSR $FFD2      bringen
130B INX            Zähler erhöhen
130C CPX #$09       schon alle Bytes?
130E BNE $1305      verzweige, wenn nicht
1310 LDX #$00       Zähler auf Null
1312 JSR $FFCF      Byte von der Tastatur holen
1315 STA $1380,X    und speichern
1318 INX            Zähler erhöhen
1319 CMP #$0D       auf RETURN warten
131B BNE $1312      verzweige, wenn noch nicht gedrückt
131D DEX            RETURN-Code abschneiden
131E STX $02        Zähler speichern
1320 LDX #$00       Password
1322 LDA $136C,X    Decodieren
1325 EOR #$24
1327 STA $136C,X
132A INX
132B CPX #$06
132D BNE $1322
132F LDX #$00       Zähler auf Null
1331 LDA $1380,X    Eingegebenes Password
```

```
1334 CMP $136C,X     mit vorgegebenem vergleichen
1337 BEQ $133C       verzweige, wenn richtig
1339 JMP $1351       ansonsten RESET
133C INX             Zähler erhöhen
133D CPX $02         vergleich mit Länge des Passwortes
133F BNE $1331       verzweige, wenn Länge nicht erreicht
1341 LDX #$00        Password
1343 LDA $136C,X     wieder
1346 EOR #$24        codieren
1348 STA $136C,X
134B INX
134C CPX #$06
134E BNE $1343
1350 RTS             Rücksprung
1351 LDX #$00        Password wieder codieren
1353 LDA $136C,X
1356 EOR #$24
1358 STA $136C,X
135B INX
135C CPX #$06
135E BNE $1353
1360 JMP $FF3D       RESET
```

Bildschirmtext und Passwort:

```
1363 50 41 53 53 57 4F 52 54 PASSWORD
136B 3A 6C 65 76 65 68 60 00 :leveh'.
1373 00 68 60 00 00 00 00 00 .h'.....
137B 00 00 00 00 00 00 00 00 ........
```

Nachfolgend der BASIC-Loader:

```
100 FORI=1TO120STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE4863+I+J,A:NEXT:READA: IFC=ATHENC=0:NEXT:
END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
```

```
300 DATA20,42,C1,A2,00,BD,63,13,20,D2,FF,E8,E0,09,D0, 138
301 DATAF5,A2,00,20,CF,FF,9D,80,13,E8,C9,0D,D0,F5,CA, 2
302 DATA86,02,A2,00,BD,6C,13,49,24,9D,6C,13,E8,E0,06, 189
303 DATAD0,F3,A2,00,BD,80,13,DD,6C,13,F0,03,4C,51,13, 180
304 DATAE8,E4,02,D0,F0,A2,00,BD,6C,13,49,24,9D,6C,13, 245
305 DATAE8,E0,06,D0,F3,60,A2,00,BD,6C,13,49,24,9D,6C, 69
306 DATA13,E8,E0,06,D0,F3,4C,3D,FF,50,41,53,53,57,4F, 9
307 DATA52,54,3A,6C,65,76,65,68,60,00,00,68,60,00,00, 28
308 DATA00,00,00,00,00,00,00,00,00,00,00,00,00,00,00, 0
```

Pass 3

```
1300 SEI            Interrupt verhindern
1301 LDA #$00       Port A
1303 STA $DC00      auf LOW setzen
1306 LDA $DC01      Port B holen
1309 CMP #$FF       Taste gedrückt
130B BEQ $1301      verzweige, wenn nicht gedrückt
130D LDX #$00       Zähler auf Null
130F LDA #$FE       Alle Bits bis auf das erste setzen
1311 PHA            Wert speichern
1312 STA $DC00      an Port A übergeben
1315 LDA $DC01      Wert von Port B holen
1318 CMP $132D,X    Mit Wert aus der Tabelle vergleichen
131B BEQ $1321      verzweige, wenn Wert identisch
131D PLA            ansonsten Stapel rücksetzen
131E CLC            und zum Anfang
131F BCC $1301      verzweigen
1321 PLA            gespeicherten Wert holen
1322 INX            Zähler erhöhen
1323 CPX #$08       mit Endwert vergleichen
1325 BEQ $132B      verzweige, wenn gleich
1327 SEC            Carry setzen
1328 ROL            gelöschtes Bit um eins verschieben
1329 BNE $1311      unbedingter Sprung
132B CLI            Interrupt wieder ermöglichen
132C RTS            Rücksprung
```

Nachfolgend der BASIC-Loader:

```
100 FORI=1TO56STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE4863+I+J,A:NEXT:READA: IFC=ATHENC=0:NEXT:
END
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,A9,00,8D,00,DC,AD,01,DC,C9,FF,F0,F4,A2,00, 98
301 DATAA9,FE,48,8D,00,DC,AD,01,DC,DD,2D,13,F0,04,68, 91
302 DATA18,90,E0,68,E8,E0,08,F0,04,38,2A,D0,E6,58,60, 132
303 DATAFF,FF,FB,F7,EF,FF,FF,FF,00,00,00,01,08,00,00, 229
```

Zu Kapitel 4:

Einige Schutzmethoden, die auf dem C64 einwandfrei funkionieren, laufen leider auf dem 128er nicht, weil das Directory nicht wie beim C64 in den BASIC-Speicher, sondern direkt in den Bildschirmspeicher geladen wird.

Deshalb werden wir nur auf diejenigen Schutzmethoden zu sprechen kommen, die funktionsfähig sind. Die Schutzmethoden, die nicht erwähnt werden, laufen nicht auf dem 128er. Die funktionierenden Schutzmethoden bedürfen keiner weiteren Erklärung, da sie identisch mit denen des C64 sind.

Hier nun die Auflistung der funkionierenden Schutzmethoden:

Kapitel 4.1 Versteckter Filename
Kapitel 4.2 Verstecken der Filenamen durch Steuerzeichen
Kapitel 4.7 Gelöschte Files

zu Kapitel 5: Kopierschutz mit der Datasette

Die Datasette ist als Speichermedium beim C128 dermaßen unpopulär, daß es sich wohl kaum lohnen würde, komplexe Kopierschutzmaßnahmen für Bandaufzeichnungen zu beschreiben. Für Interessierte sei aber gesagt, daß die Ansteuerung des Datenrekorders hier genauso vor sich geht wie beim C64. Es ist dabei lediglich zu beachten, daß die Speicheraufteilung des 128er sich von der des C64 stark unterscheidet. Näheres dazu finden Sie beispielsweise in dem DATA-BECKER-Buch "128 Intern".

Zu Kapitel 6: Diskettenkopierschutz

Diskettenkopierschutz auf dem C128. Wie sieht das in der Praxis aus? Um diese Frage zu beantworten, sehen wir uns das Diskettenlaufwerk des C128 (1570/1571) einmal genauer an.

Bei dieser näheren Betrachtung stellen wir fest, daß sich das Laufwerk, verglichen mit dem Laufwerk des C64 nicht sehr verändert hat. Die 1570/71 hat schnellere Busübertragungsroutinen, ein paar "Macken", die auch in der 1541 zu finden sind, und darüber hinaus noch weitere nette Ungereimtheiten, die es in der 1541 zum Glück nicht gibt. Beide Diskettenlaufwerke verfügen über den gleichen Controller, wodurch es theoretisch möglich ist, alle in diesem Buch gezeigten Kopierschutzverfahren für den C128 zu übernehmen.

Das einzige Problem bei dieser Übernahme sind die zu ändernden DOS-Routinen-Einsprünge und die Tatsache, daß die Byte-Ready-Leitung nicht über das Overflow-Flag, sondern über Bit 8 der Adresse $180F der VIA1 abgefragt werden muß. Aufgrund dieser Umstände haben wir uns zu einer recht einfachen, jedoch bestimmt nicht schlechten Lösung entschlossen, um den C128-Besitzern die Möglichkeit zu geben, alle unsere Kopierschutzverfahren anzuwenden.

Das Zauberwort heißt Umschalten. Anders ausgedrückt bedeutet dies, daß Sie einfach vor dem Auftragen oder Abfragen des Ko-

pierschutzes die Floppy auf den 1541-Modus herunterschalten. Nach diesem Vorgang kann der Schutz ohne Schwierigkeiten verwendet werden. Nach der Abfrage können Sie wieder in den 1570/71-Modus zurückschalten. Die folgenden Befehle dienen zum Umschalten in die verschiedenen Modi.

```
OPEN 1,8,15,"U0>:M0"  =  Einschalten des 1541-Modus
OPEN 1,8,15,"U0>M1"   =  Zurückschalten in den 1570/71-Modus
```

Selbstverständlich ist es auch möglich, auf den jeweiligen Modus erst in der Floppy umzuschalten. Zu diesem Zweck müssen Sie die in der Floppy auszuführenden Programme um den entsprechenden Unterprogrammaufruf erweitern. Die Einsprünge dieser Routinen liegen folgendermaßen:

```
JSR $9032 ; Umschalten auf 1541-Mode
JSR $904E ; Umschalten auf 1571-Mode
```

Einige Kopierschutzauftrageprogramme arbeiten leider nur auf dem C64, weshalb es nötig ist, beim Auftragen des Schutzes in den C64-Modus umzuschalten. Die Abfrage funktioniert jedoch fehlerfrei auch im C128-Modus. Darunter fallen die Programme aus den Kapiteln 6.2.4 (Änderung der Block-Header) und 6.6.3 (Der 3000-SYNC-Schutz).

Zu Kapitel 7: Wie man sich gegen Knackmodule schützt

Es ist in Kürze damit zu rechnen, daß die ersten Knackmodule und -Betriebssysteme für den 128er erscheinen. Prinzipiell kann man hier die gleichen Schutzmethoden anwenden wie beim C64. Abweichungen ergeben sich nur durch die andere Speicherverwaltung.

Zu Kapitel 7.1:

Speicherstellen, die bei einem RESET gelöscht werden:

RAM-Bank 0:

```
$0000-$0101 =     0 -  257  Systemadressen
$0200-$0AC5 =   512 - 2048  Systemadressen und Bildschirm
$1000-$12FC =               Systemadressen
```

I/O-Bank: ("BANK 15")

```
$D000-$DD0F = 53248-56591  I/O-Bereich
```

Alle RAM-Bänke:

```
$FF05-$FF44 =              IRQ/NMI-Routinen
$FFFA-$FFFF =              IRQ/NMI/RESET-Vektoren
```

Zu Kapitel 7.2:

Die Methode, Daten im Floppy-Speicher abzulegen, funktioniert unverändert auf dem C128. Lediglich die Anzahl der freien Speicherstellen, die das Betriebssystem unbenutzt läßt, hat sich verringert. Übrig geblieben sind folgende Adressen:

Hexadezimal	Dezimal
0005	5
0010 / 0011	16 / 17
0014 / 0015	20 / 21
001D	29
001F	31
0021	33
0100	256
0102 / 0103	258 / 259

Das Programm zur Abfrage der Hüllkurve der Stimme drei des SIDs braucht nicht geändert zu werden. Nur hinsichtlich des "BANK"-Befehls sollten Sie das in der Einleitung gesagte beachten. Im Zweifelsfalle ist es also besser, vor der Abfrage ein "BANK 15" auszuführen.

Der Test auf die Alarmzeit der Echtzeituhr wird besser mit der zweiten CIA durchgeführt, da Betriebssystem und BASIC-Interpreter trotz Abschalten des Timer-Interrupts auf das Interrupt-Kontrollregister der CIA eins zurückgreifen. Hier nun also die geänderten Programme:

Setzen der Alarmzeit:

```
5 BANK 15:REM FALLS NOTWENDIG: I/O-BEREICH ANSPRECHEN
10 POKE56335+256,PEEK(56335+256)OR128:REM BIT 7 SETZEN
20 POKE56331+256,128+7:REM 7 UHR NACHMITTAGS
30 POKE56330+256,3*16+5:REM 35 MINUTEN
40 POKE56329+256,1*16+1:REM 11 SEKUNDEN
50 POKE56328+256,1:REM 1 ZEHNTELSEKUNDE
```

Test der Alarmzeit:

```
5 BANK 15:REM FALLS NOTWENDIG: I/O-BEREICH ANSPRECHEN
10 POKE56335+256,PEEK(56335+256)AND127:REM BIT 7 LEOSCHEN
20 POKE56331+256,128+7:REM 7 UHR NACHMITTAGS
30 POKE56330+256,3*16+5:REM 35 MINUTEN
40 POKE56329+256,1*16+1:REM 11 SEKUNDEN
50 POKE56328+256,0:REM 0 ZEHNTELSEKUNDEN
70 FORI=1TO400:NEXT:REM ALARM ABWARTEN
80 A=PEEK(56333+256):REM ALARM TESTEN
90 IF (A AND 4)<>4 THEN PRINT"VERFLIXT, EIN KNACKMODUL!"
```

Zu Kapitel 8:

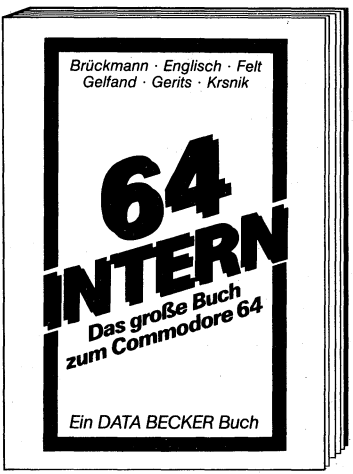
Alles, was zum Thema "rechnerzerstörender Kopierschutz" zum C64 gesagt wurde, trifft auch auf den 128er zu. Die in dieses

Gerät neu eingebauten Chips lassen keinen Ansatz für "Killer-Programme" erkennnen. Zudem sind sogar alle wichtigen Bausteine durch Wärmeableitplatten vor Überhitzung geschützt. Trotzdem geben wir auch hier den Hinweis: Falls wir uns irren, wären wir für eine Mitteilung dankbar.

Viren treffen auch im 128er-Modus auf dieselben Schwierigkeiten, die schon für den C64 beschrieben wurden. Höchstens im CP/M-Modus ließe sich vielleicht eine Möglichkeit finden, da hier das Betriebssystem von der Diskette geladen wird. Aber auch hier läßt sich durch die Verwendung von Schreibschutzetiketten jedes Virus bekämpfen.

Index

64 Intern ist ein Standardwerk zum Commodore 64, das vom ausführlich dokumentierten ROM-Listing über die detaillierte Hardwarebeschreibung bis zu nützlichen BASIC-Erweiterungen alles enthält, was man zum professionellen Einsatz des Commodore 64 wissen muß.



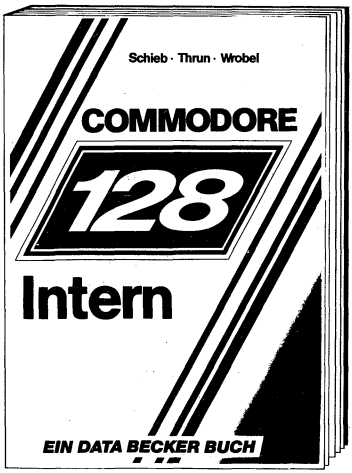
Aus dem Inhalt:

- Speicherbelegungspläne
- Der Soundcontroller und seine Programmierung
- Die Handhabung des AD-Wandlers
- Der Videocontroller
- Programmierung von Farbe und Grafik
- Die Zeichengenerator-Schnittstelle
- Sprites
- Ein-/Ausgabesteuerung
- Timer und Echtzeituhr
- Joystickprogrammierung
- So arbeitet der BASIC-Interpreter
- Mathematische Routinen – selbst entwickelt
- Der serielle IEC-Bus
- Programmierung der RS-232
- Die Belegung der Zero-Page
- Der Commodore-64-Schaltplan

Brückmann, Englisch, Felt, Gelfand, Krsnik, Gerits
64 Intern
Hardcover, 628 Seiten, DM 69,–
ISBN 3-89011-000-2

Bücher zu COMMODORE 128

Ein Standardwerk zum COMMODORE 128, das für jeden unentbehrlich ist, der tiefer in den COMMODORE 128 einsteigen will. Das gesamte Betriebssystem ist ausführlich und gründlich kommentiert, Grafik, Soundbausteine, Prozessor und Peripherieanschlüsse sind genauestens beschrieben. Ein Buch, das für den professionellen Programmierer sehr schnell unentbehrlich wird.



Aus dem Inhalt:

- Der VIC-Chip
  Registerbelegung
  Betriebsarten
  Zeichendarstellung, Graphik, Sprites und Soft-Scrolling
- Ein- und Ausgabesteuerung
  Die CIAs im COMMODORE 128
  Echtzeituhr
  Der serielle IEC-Bus des COMMODORE 128
- Der Sound-Chip SID
- Der 8563-VDC-Chip
  Pinbelegung
  Nutzung der VDC-Register
  Hires-Graphik mit 640x200 Punkten
- Das Memory-Management, die MMU
- Assemblerprogrammierung (Nutzung der Kernal-Routinen)
- Einbinden neuer BASIC-Befehle
- BASIC-Tokens
- Die CPU 8502
- Zeilenweise dokumentiertes Kernal-ROM
- ausführlich dokumentiertes BASIC 7.0
- Z-80-ROM dokumentiert (Boot-Sektor)
- Betriebssystem und Monitorlisting
- Die Hardware

Schieb, Thrun, Wrobel
128 Intern
846 Seiten, DM 69,–
ISBN 3-89011-098-3

Das große Floppybuch zur 1570/1571 gibt Ihnen das notwendige Wissen zur Programmierung Ihres neuen Diskettenlaufwerkes. Für Anfänger, Fortgeschrittene und Profis. Dieses Buch beschreibt wirklich alle Leistungsmerkmale dieser schnellen Floppy.

Aus dem Inhalt:

- Einführung für Einsteiger
- Die Floppy und das COMMODORE-BASIC
- Sequentielle und relative Dateien
- Fremde Diskettenformate verarbeiten
- Programmierung im DOS-Puffer
- Die CP/M-Fähigkeiten der 1570/1571
- Floppy intern: Schaltungsaufbau und Funktion
- 1571 Fast-Load
- Das DOS im Detail
- Komplettes DOS-Listing (mit Cross-Reference)

Ellinger
Das große Floppybuch zur 1570/1571
Hardcover, 554 Seiten, DM 49,–
ISBN 3-89011-124-6

Endlich ist es soweit: Maschinensprache für Einsteiger ist das Buch, auf das alle, die sich schon immer für Maschinensprache interessierten, gewartet haben. Finden Sie heraus, was in Ihnen und in Ihrem Rechner steckt. Dieses Buch zeigt Ihnen, wie's geht: ohne Fachchinesisch, dafür einfach, schnell und effektiv. Nutzen Sie diese Chance.

Aus dem Inhalt:

- Einführung in Assembler – Was man über Zahlen, Speicher und den Rechner wissen sollte
- Wie man mit einem Monitor arbeitet (Laden, Starten, Eingeben und Speichern eigener Programme)
- Die ersten Befehle und die Adressierungsarten
- Fortgeschrittene Assemblerprogrammierung
- Noch mehr über Schleifen
- Rechnen in Maschinensprache
- Vergleichsbefehle
- Stapeloperationen
- Interruptprogrammierung
- Die verschiedenen Rechner/Programmierhilfsmittel für C16, 116, Plus/4 und C128
- Und viele, viele Beispiele ...

Baloui
Maschinensprache für Einsteiger
346 Seiten, DM 29,–
ISBN 3-89011-182-3

DAS STEHT DRIN:

Vom Einsteiger bis zum Softwarehaus – dieses Buch zeigt, wie man seine Programme optimal schützt. Dabei werden Programm- und Kopierschutzverfahren vom einfachen List-Schutz bis zur Überprüfung eines kompletten Tracks gezeigt. Und: Die Autoren sind nicht bei bekannten Schutzmechanismen der neuesten Generation stehengeblieben, sondern haben neue entwickelt, für die es bisher keine Kopiermöglichkeiten gibt. Das Buch zeigt nicht nur die Verfahren, sondern liefert auch gleich fertige Lösungen für C64 und C128 zum Abtippen.

Aus dem Inhalt:

- BASIC-Programme schützen (List-Schutz, Änderungs-Schutz, Password, RESTORE/RESET-Schutz, SYS-Bluff, Compiler-Schutz)
- Alle Tricks des Autostart (Tastaturpuffer, Sprungvektoren, Stack, Interrupt, Adreßverschiebung beim Laden)
- Prüfsummen und Selbstzerstörung (XOR-Codierung, Timer-Codierung, Einzelschrittcodierung, ASCII-Codierung)
- Illegale Opcodes
- Cassettenkopierschutz (EOF, Cassettenpuffer, Autostart, Schnelladeverfahren, komplettes Kopierschutzsystem)
- Directory-Schutz (versteckter Filename, "blinde" Blöcke, der Nullen-Trick, geteiltes Directory, gelöschte Files)
- Grundlagen des Kopierschutzes
- Alle Tricks der Formatierung (Track 35-41, einzelne Tracks, doppelte Tracks, Header-Parameter ändern, zerstörte Sektoren, gleiche Sektoren auf einem Track)
- Halftracks, der Track mit den Nullen, kompletten Track prüfen
- Alle Tricks der SYNC-Markierungen (Killertracks, überlange SYNC-Markierungen, 3000-SYNC-Schutz, Daten ohne SYNC-Markierungen)
- Der Disketten-Killer (nur Ausschalten der Floppy hilft)
- Alle Tricks der Knack-Module und wie man sich dagegen schützt
- Was Software-Häuser über Cracker wissen sollten

UND GESCHRIEBEN HABEN DIESES BUCH:

Jacques Felt und Darko Krsnik sind Hardware- und Assemblerspezialisten. Ralf Gelfand ist absoluter Floppy-Experte. Er hat schon Kopierschutzverfahren für den professionellen Einsatz entwickelt. Von Michael Strauch stammt das Kapitel über die Datasette, die er bis zum letzten Byte kennt.

ISB N 3-89011-253-6 DM +039.00

öS 304,–
sFr 37,–

03900

9 783890 112534

DATA BECKER