Memory Map mit Wandervorschlägen (10)

Diesmal geht es weiter mit den Speicherzellen 158 bis 182. Behandelt werden dabei die interne Uhr, Kassettenoperationen und die RS232-Schnittstelle.

In der Ausgabe 6/85 habe ich unter anderem auch die Speicherzelle 144 behandelt. In einem separaten Texteinschub »ST-ATUS« bin ich näher auf die Status-Variable ST eingegangen.

Ganz am Schluß, nach der Besprechung der Floppy-Operationen, habe ich noch den Wert ST=128 (DEVICE NOT PRESENT) behandelt.

Ich mußte allerdings eingestehen, daß mir nicht bekannt war, wie man den Wert 128 innerhalb eines Programms abfragen könnte, um eine Programmunterbrechung bei abgeschalteten Peripherie-Geräten zu vermeiden.

Dieses Eingeständnis hatte unerwartete Folgen. Ich habe mehrere Zuschriften erhalten, die mir Hinweise gaben, wie das Problem zu lösen ist:

— Herr Witte aus Wunstorf
— Herr Bojko aus Dortmund
— Herr Stahl aus Stuttgart
— Herr Kühlewind aus Berchtesgaden
— Herr Möller und Dipl.-Ing. Minning aus Marburg
— Herr Müller aus Wesel

Ich muß ehrlich sagen, daß ich mich über soviel »Mitarbeit« sehr gefreut habe. Allen Schreibern danke ich hiermit herzlich für die Hinweise.

Ich kann hier nicht alle Lösungen vorstellen. Es ist auch nicht notwendig, da sie alle sehr ähnlich sind. Ich habe deswegen eine Zusammenfassung erstellt, die — wie ich hoffe — allen Beiträgen gerecht wird.

Es gibt zwei Speicherzellen — 768/769 —, auf die wir bei unserer Wanderung durch die Speicherlandschaft noch kommen werden, in denen in Low-/High-Byte-Darstellung eine Adresse steht, auf die das Betriebssystem springt, wenn die Meldung »DEVICE NOT PRESENT« ausgegeben werden soll. Diesen Zeiger kann man so »verbiegen«, daß die Meldung nicht ausgegeben wird, und daß das Programm einfach weiterläuft.

Normalerweise steht in 768 die Zahl 139 (VC 20: 58), in 769 die Zahl 227.

Verbogen wird der Zeiger durch eine 61 (185 geht auch), beim VC 20 durch 52. Dadurch zeigt die Adresse auf eine Speicherzelle des Betriebssystems, in welcher der Assembler-Befehl »RTS«, das bedeutet Rücksprung, steht. Jetzt können wir ungestört den STatus abfragen, wir müssen allerdings den negativen Wert von ST, also —128 nehmen.

Für die Floppy sieht die Abfrage fast gleich aus.

Die einzige Änderung ist in den Zeilen 20 und 30:
20 OPEN 1,8,15
30 diese Zeile entfällt.

Unter bestimmten Umständen kann das Verbiegen des Zeigers in 768 entfallen, wie einige Zuschriften ergeben haben.

Ich möchte nach eigenen längeren Versuchen aber dafür plädieren, die Fehlermeldung immer abzuschalten, um nie in Schwierigkeiten zu kommen.

Vorsicht ist die Mutter der Weisheit.

Für den Drucker sieht das so aus:

```
 10 POKE 768,61                  Fehlermeldung abschalten
 20 OPEN 1,4                     Gerät ansprechen
 30 PRINT #1
 40 CLOSE 1
 50 POKE 768,139                 Fehlermeldung einschalten
 60 IF ST=—128 THEN 100          Sprung bei ausgeschaltetem
                                 Gerät
 70 PRINT" FORTSETZUNG"          Weiter im Programm
 80 END                          Ende der DEMO
100 PRINT" GERAET EINSCHALTEN"
110 GET A$:IF A$=""THEN 110      Warteschleife
120 GOTO 10                      neuer Versuch
```

Adresse 158 und 159 ($9E und $9F)

Zwischenspeicher bei Kassettenoperationen

Diese beiden Speicherzellen werden von Routinen des Betriebssystems verwendet, welche bei Kassettenoperationen die Zeichen überprüfen, ob sie richtig sind, und welche bei aufgetretenen Fehlern Korrekturen durchführen.

Adresse 160 bis 162 ($A0 bis $A2)

Interne Uhr für TI und TI$

Das Basic der Commodore-Computer kennt neben der Variablen ST (siehe Speicherzelle 144) noch zwei weitere »reservierte« Variable, nämlich TI und TI$. Beide bieten eine interne Uhr, welche aus dem Inhalt der Speicherzellen 160 bis 162 abgeleitet wird. Diese drei Zellen funktionieren wie der Kilometerzähler eines Autos, halt nur mit drei Stellen.

Die hinterste Stelle ist die Zelle 162. Ihr Inhalt wird beim Einschalten des Computers auf 0 gesetzt, dann aber 60mal in der Sekunde um 1 erhöht. Das erfolgt durch die automatische Interrupt-Routine, welche auch die STOP-Taste abfragt und noch andere Hausaufgaben 60mal in der Sekunde ausführt. Da 1/60 = 0,01667 ist, zählt also die Zelle 162 in 0,01667 Sekunden um 1 weiter. Sie kann wie alle Speicherzellen maximal nur die Zahl 255 enthalten, danach kommt wieder eine 0. Das heißt aber, daß sie nach 256 * 0,01667 = 4,267 Sekunden einmal durchgelaufen ist.

Nach jedem Durchlauf wird die davorliegende Speicherzelle 161 um 1 erhöht. Sie zählt also in 4,267 Sekunden um 1 weiter und ist nach 256 * 4,067 = 1092,26 Sekunden oder besser nach 18,2044 Minuten einmal durchgelaufen. Nach dem Kilometerzähler-Prinzip wird nach jedem Durchlauf von 161 der Inhalt der davorliegenden Zelle 160 um 1 erhöht.

Die Zelle 160 zählt also in 18,2044 Minuten um 1 weiter und ist nach 256 * 18, 2044 = 4660,34 Minuten, das sind 77,67 Stunden, einmal durchgelaufen.

Diese Stundenzahl wird allerdings niemals erreicht, da das Betriebssystem nach Erreichen des Wertes für 24 Stunden alle drei Zellen wieder auf 0 zurücksetzt. Wir werden das gleich nachprüfen.

Zuerst aber wollen wir uns den dreizelligen Zähler anschauen:

```
10 PRINT PEEK(160);PEEK(161);
PEEK(162)
20 GOTO 10
```

Nach RUN sehen wir den Inhalt der drei Zellen sich entsprechend der oben angegebenen Zeiten verändern. Die Zahlen sind nicht vorherbestimmbar, denn der Zähler ist ja nach dem Einschalten des Computers schon losgelaufen. Er kann aber auf 0 gesetzt werden durch Einfügen der Zeile 5:

```
5 POKE 160,0:POKE 161,0:POKE
162,0
```

Jetzt beginnt der Zähler immer ab 0. Ich habe gerade gesagt, daß der Zähler auf 0 gesetzt wird, wenn er 24 Stunden lang gelaufen ist. Der Inhalt in den drei Speicherzellen, der 24 Stunden entspricht, ist nach der oben angegebenen Umrechnungsart 79-26-0. Diesen Wert, oder besser noch ein Wert kurz davor, in die Zellen 160 bis 162 gePOKEt, zeigt uns den Nullsetzvorgang. Ersetzen Sie bitte die obige Zeile 5 durch eine neue Zeile:

```
5 POKE 160,79:POKE 161,25:
POKE 162,0
```

Nach vier Sekunden Laufzeit schalten alle drei Zellen in der Tat auf 0 zurück.

Die Umsetzung der Zahlen aus 160 bis 162 in die Variablen TI und TI$ sowie deren Wirkungsweise, entnehmen Sie bitte dem nebenstehenden Texteinschub 1 »Die eingebaute Uhr«.

Abschließend muß eines noch warnend erwähnt werden. Alle Operationen, welche den Interrupt-Vektor verwenden beziehungsweise verändern, stören oder verzögern die normale Interrupt-Routine, die ja den Zähler weiterstellt. So zählt der Zähler nicht gleichmäßig und die daraus abgeleitete Uhr geht nicht mehr richtig. Ein Beispiel dafür sind alle Ein- und Ausgaben über die Datasette, welche über einen Interrupt laufen.

Adresse 163 bis 164 ($A3 und $A4)

Zwischenspeicher

Diese beiden Speicherzellen werden von den Ein- und Ausgabe-Routinen des Betriebssystems für Kassetten, Floppy-Laufwerk und Drucker als Zwischenspeicher für alle möglichen Werte benutzt.

Adresse 165 ($A5)

Bit-Zähler für Kassetten-Synchronisierung

Beim Abspeichern eines Programms auf ein Band werden vor den eigentlichen Daten mehrere Bits zusätzlich gespeichert, die beim Einlesen dieses Bandes zur Synchronisierung dienen, das heißt zum Übereinstimmen der Geschwindigkeit der Datenübertragung.

Die Speicherzelle 165 wird als Zähler dieses Synchron-Bits verwendet.

Adresse 166 ($A6)

Zähler der bearbeiteten Bytes im Kassetten-Puffer

Diese Speicherzelle wird als Zähler benutzt, welcher angibt, wieviele Bytes gerade in den Kassetten-Puffer eingeschrieben oder aus ihm ausgelesen worden sind. Der Kassetten-Puffer besteht aus den Speicherzellen 828 bis 1019 und kann somit 191 Byte aufnehmen, was zugleich die höchste Zahl ist, welche sinnvollerweise in der Zelle 166 stehen kann.

Nähere Erklärungen und ein paar Experimente mit Zelle 166 finden Sie in dem nebenstehenden Texteinschub 2 »Experimente mit dem Kassetten-Puffer«.

Die meisten der nächsten 20 Speicherzellen werden bei Operationen mit der RS232-Schnittstelle, die über den User-Port den Computer mit anderen Geräten verbindet, eingesetzt. Da die Programmierung der RS232-Schnittstelle nicht zu den üblichen Programmierarbeiten zählt, sondern nur von wenigen Spezialisten verwendet wird, gehe ich auf diese RS232-Adressen nicht im Detail ein. Dies sollte einem eigenen Spezialkurs vorbehalten sein.

Adresse 167 ($A7)

Zwischenspeicher für Kassetten-Operationen und für Eingabe über die RS232-Schnittstelle

Diese Speicherzelle wird verwendet, um jedes Bit, welches von einem RS232-Kanal über den User-Port eingelesen wird, zwischenzuspeichern.

Außerdem verwenden mehrere Kassetten-Routinen diese Adresse als Zwischenspeicher.

Adresse 168 ($A8)

Bitzähler für RS232-Eingabe und bei Band-Ein-/Ausgabe

Die Speicherzelle 168 wird als Zähler verwendet, der diesmal nicht die Bytes, sondern die Anzahl der Bits zählt, die sowohl über den User-Port als auch über den Kassetten-Port geleitet werden. Das dient dem Betriebssystem dazu, zu wissen, wann ein volles Wort abgearbeitet worden ist.

Adresse 169 ($A9)

RS232-Flagge für Startbit-Prüfung

Ein RS232-Datentransfer prüft, ob ein Start-Bit empfangen worden ist. Im positiven Fall steht in Zelle 169 die Zahl 144, im negativen Fall eine 0.

Adresse 170 ($AA)

RS232-Eingabe- und Zwischenspeicher für Kassetten-Routinen

Bei der Speicherzelle 165 haben wir gesehen, daß ein Band Synchronisationsbits enthält. Die Speicherzelle 170 wird dabei als Flagge benutzt, die angibt, ob ein gelesenes Zeichen Synchronisierungs-Bits oder ein Datenwort darstellt.

Die RS232-Routinen verwenden Zelle 170 dagegen als Speicher, in welchem die eingelesenen Bits zu einem Byte zusammengefaßt werden, bevor sie im Eingabepuffer am oberen Ende des Programmspeichers abgelegt werden (siehe auch Speicherzellen 55/56).

Adresse 171 ($AB)

Quersummenprüfung und Zähler für Band-Header bei RS232- und Kassetten-Operationen

Diese Speicherzelle wird vom Betriebssystem benutzt um festzustellen, ob während einer RS232-Datenübertragung Bits verloren wurden. Da derartige Prüfungen mit Parity-Bits (Quersummenprüfung) des öfteren erwähnt werden, gebe ich eine kurze Beschreibung des Prüfprinzips im nebenstehenden Texteinschub 3 »Fehlererkennung mit Parity-Bits«.

Zusätzlich wird in 171 die Länge des Band-Vorspanns bei seiner Erzeugung gezählt.

Adresse 172 und 173 ($AC und $AD)

Zeiger auf die Anfangsadresse für Ein-/Ausgabe, Zwischenspeicher für den Bildschirmeditor

In den Speicherzellen 193/194 steht ein Zeiger, der auf die Adresse im Programmspeicher zeigt, wo das Programm beginnt beziehungsweise beginnen soll, welches abgespeichert beziehungsweise geladen werden soll.

Dieser Zeiger wird am Anfang einer Lade- oder Abspeicher-Operation in die Zellen 172/173 gebracht, wo er während der Operation laufend erhöht wird, bis das Ende des Programms erreicht ist; dann wird er wieder auf seinen ursprünglichen Wert gesetzt.

Der Zeiger dient außerdem noch dem Bildschirmeditor als Zwischenspeicher während des Scrollens (Hochschieben) des Bildschirms und beim Einfügen zusätzlicher Zeilen.

Dieser Zeiger kann sehr nützlich sein, um Programme entweder schon beim SAVEn oder aber erst beim LOADen gezielt auf andere als ursprüglich verwendete Speicherbereiche zu bringen. Dazu sind aber noch einige andere Zellen notwendig, bis hin zu dem schon erwähnten Zeiger in 193/194. Ich werde mit dieser Anwendung und ihrer Beschreibung daher warten, bis wir zu 193/194 kommen.

Adresse 174 und 175 ($AE und $AF)

Zeiger auf die Endadresse für Ein-/Ausgabe, Zwischenspeicher für den Bildschirmeditor

Dieser Zeiger ist der Zwilling zu 172/173, nur zeigt er seinerseits auf die letzte Adresse des zu bewegenden Programms (siehe oben).

Adresse 176 und 177 ($B0 und $B1)

Zeitkonstante

Der Wert in dieser Speicherzelle wird verwendet, um die Zeitkonstante zum Lesen vom Band in der Zelle 146 einzustellen.

Adresse 178 und 179 ($B2 und $B3)

Zeiger auf den Kassetten-Puffer

Beim Einschalten des Computers werden diese Speicherzellen in Low-/High-Byte-Darstellung auf die Anfangsadresse des Kassetten-Puffers gesetzt. Beim VC 20 und C 64 ist dies die Adresse 828 ($33C).

Adresse 180 ($B4)

RS232-Bit-Zähler und -Zwischenspeicher für Kassetten-Operationen

Die RS232-Routinen verwenden die Speicherzelle 180, um die Zahl der übertragenen Bits zu zählen, außerdem für Parity-Berechnung (siehe Texteinschub 3) und Stop-Bit-Bearbeitung. Die Lade-Routinen für Kassettenbetrieb benutzen diese Zelle als Flagge, die angibt, ob der Computer bereit ist, Daten zu übernehmen.

Adresse 181 ($B5)

RS232-Anzeige für nächstes Bit, Flagge für End-of-Tape

Bei RS232-Operationen enthält die Zelle 181 das jeweils nächste Bit, welches übertragen werden soll. Bandoperationen entnehmen dieser Speicherzelle, welcher Block gerade gelesen wird.

Adresse 182 ($B6)

Ausgabe-Zwischenspeicher für RS232 und Kassette

Bei Ausgabe von Daten über die RS232-Schnittstelle wird jedes Byte in seine Einzelteile zerlegt, bevor es über den Ausgabepuffer seriell übertragen wird. Der Ausgabepuffer wird im obersten Teil des Programmspeichers angelegt (siehe auch Speicherzellen 55 und 56); die genaue Anfangsadresse steht in Speicherzelle 248. Auch die Ausgabe von Daten auf die Kassette verwendet Zelle 182 als Ausgabe-Zwischenspeicher.

(Dr. H. Hauck/ah)

Texteinschub # 1 Die eingebaute Uhr

Der VC 20 und der C 64 haben eine interne Uhr eingebaut, deren Stand abgefragt, ausgedruckt und somit zur Zeitmessung und Programmsteuerung eingesetzt werden kann.

In der Basic-Befehlsliste der Handbücher finden wir dazu zwei Funktionen, TI und TI$.

1) TI gibt den Stand des Zählers wieder, der durch die drei Speicherzellen 160, 161 und 162 gebildet wird. Dabei ist der Wert von TI nichts anderes als die Summierung des Inhalts dieser drei Zähler.

Entsprechend dem dreistelligen Zählerprinzp (siehe Beschreibung der Speicherzellen 160 bis 162) ist die Summe:

```
TI    = Inhalt (162)
      + Inhalt (161) * 256
      + Inhalt (160) * 256 * 256
```

Mit dem folgenden kleinen Programm können wir das verifizieren:

```
10  PRINT TI;
20  PRINT PEEK(162)+256*PEEK(161)+256*256*PEEK(160)
30  GOTO 10
```

Die beiden Zahlenbänder für TI und die Zählersumme sind praktisch identisch.

2) TI$ gibt ebenfalls den Stand des Zählers wieder, aber in einer anderen Darstellung. Während TI 60mal in der Sekunde weiterzählt, gibt TI$ direkt Stunden, Minuten und Sekunden an.

Den Zusammenhang zwischen TI und TI$ können Sie am besten mit dem folgenden kleinen Programm sehen:

```
10  PRINT INT(TI/60);
20  PRINT TI$
30  GOTO 10
```

Zeile 10 rechnet TI in Sekunden um. Damit die Zeile nicht mit vielen Dezimalstellen volläuft, verwandelt sie das Resultat in eine ganze Zahl. Zeile 20 zeigt dazu im Vergleich die sechs Ziffern von TI$.

Das erste, was beim Ablauf des Programms auffällt, ist die gleichzeitige Umschaltung beider Zahlenreihen. Die Umrechnung von TI$ nach TI geht am besten »zu Fuß«. Stoppen Sie den Lauf mit der STOP-Taste. Nehmen Sie dann den letzten Wert von

TI$ (rechts). Die ersten beiden Ziffern sind die Stunden, ihr Wert wird mit 3600 multipliziert, um sie in Sekunden umzurechnen. Addieren Sie dazu den Wert der mittleren beiden Ziffern (Minuten) multipliziert mit 60, und addieren Sie zu diesem Zwischenergebnis die Sekunden (Ziffern ganz rechts). Das Resultat ist identisch mit dem letzten Wert von TI.

Wenn Sie übrigens den Ausdruck für TI$ optisch verbessern wollen, dann setzen Sie zwischen die Stunden, Minuten und Sekunden einen Doppelpunkt. Das wird durch eine String-Manipulation erreicht:

```
Print LEFT$(TI$,2)":"MID$(TI$,3,2)":"RIGHT$(TI$,2)
```

Eine gute Uhr muß sich stellen lassen — bei TI$ erreichen wir das einfach mit Zuweisen des gewünschten Wertes an die Variable TI$. Zum Beispiel stellt

```
TI$ = "153000"
```

die Uhr auf 15 Uhr 30. Man kann sie dementsprechend auch auf 0 zurücksetzen, was bei einem Stoppuhr-Betrieb notwendig wird.

TI kann direkt nicht beeinflußt werden, nur über POKEn von neuen Werten in die Speicherzellen 160 bis 162 oder durch die Zuweisung von Werten an TI$.

Die eleganteste Methode, TI und TI$ auf 0 zu setzen, geht beim C 64 und VC 20 mit

```
SYS 65499
```

Wenn Sie noch das kleine Programm von oben im Rechner haben, können Sie es gleich ausprobieren. Geben Sie direkt ein:

```
SYS 65499:RUN
```

und die Uhr startet von Null an.

Abschließend möchte ich Ihnen noch zwei kleine Anwendungsbeispiele von TI und TI$ mitgeben. Das erste ist ein Kochrezept, wie die Laufzeit eines Programms gemessen werden kann. Diese Programm-Stoppuhr besteht aus zwei Zeilen.

Die erste Zeile setzt die Uhr auf 0, das kennen wir schon.

Die zweite Zeile druckt am Ende des Programms die abgelaufene Zeit aus.

```
10 TI$ = "000000"
:
:
10000 PRINT TI/60 "SEKUNDEN"
```

Das zu messende Programm steht zwischen diesen beiden Zeilen.

Das zweite Beispiel betrifft eine Uhr, die nach einer vorgegebenen Zeit ein Programm (Spiel) abbricht. Davon zeige ich zwei Versionen. Die eine Version ist nach allen Erklärungen von oben beinahe trivial:

```
10 TI$ = "000000"
1000 IF TI$ >"000700" THEN STOP
```

Diese beiden Zeilen setzen die Uhr auf 0 und brechen ein Programm nach genau 7 Minuten ab.

Etwas kniffliger ist der Abbruch (oder Start) mit einer Count-Down Uhr.

```
10   TI$ = "000000"
20   ZEIT = 300
30   IF ZEIT-VAL(TI$) <=0 THEN STOP
40   weiteres Programm
```

Die Variable »Zeit« gibt die Dauer des Count-Down in Sekunden an. Zeile 30 überprüft den Wert von TI$, bis er 300 erreicht hat, indem sie den jeweiligen Wert von TI$ von der vorgegebenen Zeit subtrahiert. Natürlich müssen in beiden Versionen die Prüfzeilen sinnvoll in ein Programm eingebaut werden. Aber das möchte ich gern Ihnen überlassen.

Texteinschub # 2 Experimente mit dem Kassetten-Puffer

Die Speicherzellen von 828 bis 1019 werden als »Kassetten-Puffer« bezeichnet.

Beim Abspeichern auf eine Kassette wird zuerst der Vorspann eines Bandes, der sogenannte »Header«, in diesen Puffer gespeichert. Ein Programm wird dann direkt auf das Band geschrieben. Eine Datei allerdings läuft zuerst auch in den Kassetten-Puffer und von dort erst auf das Band. Sie kennen sicher die charakteristischen Wartezeiten des Kassettenmotors beim SAVEn einer Datei.

Beim Laden von einer Kassette gilt der Unterschied zwischen einem Programm und einer Datei genauso, einschließlich der Benutzung des Kassetten-Puffers.

Wir haben gelernt, daß in der Speicherzelle 166 die Zahl der Bytes gezählt wird, die in den Puffer geschrieben, beziehungsweise aus dem Puffer gelesen worden sind. Die Zahl reicht von 0 bis 191.

Diese Speicherzelle 166 kann während eines Programms abgefragt und auch mit POKE beliebig verändert werden. Was dabei herauskommt, ist vordergründig nur eine Spielerei. Aber vielleicht kann man die folgenden Experimente auch nutzbringend einsetzen.

Zuerst wollen wir die Funktionsweise von 166 erproben. Dazu laden wir eine simple Datei auf ein leeres Band, und zwar mit folgendem Programm:

Programm # 1

```
10   OPEN 1,1,1
20   FOR I=100 TO 150
30   PRINT #1,I
40   NEXT
50   CLOSE 1
```

Wir eröffnen eine Datei (ohne Namen) mit der Nummer 1, für Kassette (die zweite 1), zum Schreiben (die dritte 1). Nach RUN wird der Kassetten-Puffer mit den Zahlen 100 bis 150 in mehreren Schüben gefüllt, wobei jeder Schub einzeln auf das Band geschrieben wird.

Den Zusammenhang zwischen den Datei-Zahlen und dem Zähler in 166 zeigt uns das folgende Ausleseprogramm:

Programm # 2

```
10   OPEN 1,1,0
20   GET #1,X$
30   Print X$;
40   PRINT CHR$(28)PEEK(166)CHR$(154);
50   GOTO 20
```

Wir eröffnen wieder eine Datei, diesmal zum Lesen (die 0), und bringen mit GET# die einzelnen Zeichen hintereinander in den Puffer und dann auf den Bildschirm. Die Zeile 40 druckt nach jedem Zeichen in roter Farbe [CHR$(28)] den Zählerstand und schaltet dann mit CHR$(154) — beim VC 20 wäre das CHR$(31) — wieder auf die Normalfarbe zurück.

Zuerst muß das Band zurückgespult werden, und dann geht es los mit RUN. Nach dem Erscheinen der ersten Zeichen auf dem Bildschirm stoppen Sie bitte den Ablauf mit der STOP-Taste.

Sie sehen jetzt in Rot den Inhalt der Zelle 166, die aufwärts zählt, und dazwischen in Blau die Zahlen von 100 aufwärts. Interessant ist, daß durch Zwischenräume für eine 3stellige Zahl 6 Byte verbraucht werden.

Fahren Sie mit CONT solange fort, bis der Kassettenmotor anläuft und der nächste Schub auf dem Bildschirm ausgedruckt wird. Nach erneutem STOP sehen Sie, daß die roten Zahlen nach 190 wieder auf 0 zurückspringen. Das war der Moment, wo der Kassettenmotor wieder eingeschaltet wurde.

Diese Erkenntnis verwenden wir für ein Experiment.

Mit der in das Programm # 2 eingeschobenen Zeile 45 fragen wir den Inhalt von 166 ab und beeinflussen damit dem Ablauf des Programms. Außerdem setzen wir an dieser Abfragestelle den Inhalt der Zelle 166 auf den Endwert 191 und zwingen damit den Kassettenmotor weiterzulaufen.

```
45 IF PEEK(166) = 18 THEN POKE 166,191
```

Die Wiederholung des Programms mit zurückgespultem Band bringt uns ein neues Ergebnis:

Sobald der Zähler in 166 die 18 erreicht hat, glaubt das Programm, der Kassetten-Puffer wäre bereits ausgelesen, schaltet den Kassettenmotor wieder ein und liest den nächsten Zahlenblock in den Puffer. Wir erhalten jetzt nicht alle Zahlen, die auf dem Band stehen, sondern nur Gruppen von 18 Bytes, das sind ungefähr drei Zahlen.

Ich sage »ungefähr« mit Absicht, denn mit der Symmetrie, beziehungsweise mit der richtigen Reihenfolge klappt es nicht immer so ganz, da ja die Länge des Kassetten-Puffers nicht unbedingt ein ganzzahliges Vielfaches der ausgelesenen Bytes ist. Da liegt also ein kleines Problem.

Dieses Abfragen und Abändern der Speicherzelle 166 geht natürlich auch in der anderen Richtung, nämlich beim Abspeichern von Zahlen. Nehmen Sie bitte nochmal das Programm # 1 her und ändern Sie es wie folgt ab:

Programm # 1.a

```
10   OPEN 1,1,1
20   FOR I=100 TO 300
30   PRINT#1,I
35   IF PEEK(166)=18 THEN POKE 166,191
40   NEXT
50   CLOSE 1
```

Wir haben jetzt die Abfrage der Speicherzelle 166 des Programms # 2 von vorhin in das Programm # 1 eingebaut. Spulen Sie bitte das Band zurück und lassen Sie das Programm laufen.

Nun wollen wir die dadurch neu abgespeicherte Datei ganz normal auslesen. Dazu nehmen wir das Programm #2, also ohne die Zeile 45. Das sieht dann so aus:

Programm #2.a

```
10 OPEN 1,1,0
20 GET#1,X$
30 PRINT X$;
40 PRINT CHR$(28)PEEK(166)CHR$(154);
50 GOTO 20
```

Wir starten es mit RUN, nachdem das Band wieder zurückgespult ist. Der Vorgang ist im Prinzip der gleiche wie bei Programm #2; halten Sie das Programm bitte auch wieder an, so wie vorher.

Wir sehen aber einen großen Unterschied im Ausdruck. Es erscheinen nur die ersten drei Zahlen, 100 bis 102, danach steht nichts mehr im ganzen Block, bis der Inhalt von 166 die Endzahl 190 erreicht hat. Erst danach, nach dem Loslaufen des Kassettenmotors und dem Einlesen des nächsten Schubes, erscheinen die nächsten drei Zahlen.

Schlußfolgerung:

Durch POKEn der Zahl 191 in die Speicherzelle 166 zu einem beliebigen Zeitpunkt können wir sowohl beim Abspeichern, als auch beim Einlesen einer Datei dem Computer vorgaukeln, der Kassetten-Puffer sei bereits abgearbeitet. Dadurch wird der Kassettenmotor eingeschaltet und der nächste Schub ein- beziehungsweise ausgelesen.

Texteinschub #3 Fehlererkennung mit Parity-Bits

Bei der Datenübertragung zwischen Peripheriegeräten, insbesondere zwischen Datasette und dem Computer kommt es recht häufig vor, daß Fehler auftreten. Diese Fehler haben alle möglichen Ursachen und trotz aller Anstrengungen der Ingenieure lassen sie sich leider nicht völlig vermeiden.

An besonderen Schwachstellen werden daher Maßnahmen getroffen, um Fehler wenigstens zu erkennen und Programme abzubrechen, bevor größerer Schaden entsteht. Die mißlichen »LOAD ERROR«-Meldungen sprechen da eine deutliche Sprache. Die einfachste Art, Fehler zu erkennen — ich sollte genauer sagen: einzelne Bitfehler zu erkennen — geschieht über sogenannte »Parity-Bits«. Die Methode besteht darin, daß zu einem Datenwort, zum Beispiel einem Byte, ein zusätzliches Bit hinzugefügt wird und zwar so, daß die Quersumme immer eine gerade oder auch eine ungerade Zahl ergibt.

Bevor ein Wort übertragen wird, errechnet der Sender das Parity-Bit und fügt es dem Wort als zusätzliches Bit hinzu. Der Empfänger, der diese Prüfmethode natürlich auch kennen muß, rechnet die Quersumme aus. Wenn sie stimmt, nimmt er das Parity-Bit weg und arbeitet mit dem richtigen Wort weiter. Wenn die Quersumme nicht stimmt, schlägt er Alarm.

Sie werden sicher schon bemerkt haben, daß in meinem Beispiel natürlich ein Doppelfehler, nämlich zwei falsche Bits, natürlich nicht erkannt werden. Um das zu erreichen, müßte man zwei Parity-Bits einführen. Sie sehen natürlich auch, wohin das letztlich führt, nämlich zu einer Vergrößerung der Wortlänge. Man nennt das auch »Redundanz«, vielleicht haben Sie dieses Wort schon einmal gehört. Nun, da gibt es für jeden Anwendungsfall ein Optimum, abhängig von der Wahrscheinlichkeit, welche Art von Fehlern in welcher Häufigkeit auftreten. Im Extremfall gibt es Codiersysteme — zu denen die Parity-Bit-Methode auch gehört — welche in der Lage sind, Fehler nicht nur zu erkennen, sondern gleich zu korrigieren.

Logeleien (Teil 3)

Diese letzte Folge der Logeleien wird zum krönenden Abschluß unseren Commodore 64 als »Schlußfolgerungs-Maschine« vorstellen und den Zusammenhang mit der sogenannten Künstlichen Intelligenz beleuchten.

Das Aschenputtel unseres Basic-Sprachvorrates, den WAIT-Befehl, hatten wir in der letzten Folge untersucht. Dabei kam heraus, daß er — mit nur einem Argument versehen — testet, ob in einer spezifizierten Speicherstelle ein bestimmtes Bit gesetzt ist. Im Gegensatz dazu prüft WAIT mit zwei Argumenten, ob ein bestimmtes Bit gelöscht ist.

16. Nochmal WAIT:
Diesmal mit zwei Argumenten.

Wie funktioniert das? Gehen wir aus vom Befehl

WAIT C,A,B

Dabei ist C die abzufragende Speicherstelle, A eine AND-Maske und B eine EOR-Maske. Sogar im Programmers Reference Guide findet man eine falsche Funktionsbeschreibung. Tatsächlich geschieht folgendes:

1) Der Wert der Speicherstelle C wird gelesen.
2) Dieser Wert wird mit B als EOR-Maske exklusiv-oder-verknüpft
3) Das Ergebnis davon wird mit A als AND-Maske verknüpft.

Anhand eines Beispiels wollen wir uns ansehen, was passiert. Der Befehl

WAIT 1,32,32

hält ein Programm an, bis eine Datasettentaste gedrückt wird (H. Hauck, 64'er, Ausgabe 11/1984, Seite 135). Bit 4 der Speicherstelle 1 ist in dem Fall nämlich Null. Der normale Inhalt dieses Registers ist 55 (binär 0011 0111), also Bit 4 = 1. Die Zahl 32 lautet binär

0001 0000

Mit WAIT1,32,32 geschieht folgendes:

keine Taste:

```
 0011 0111    55
 0001 0000    32
_________   EOR
 0010 0111
 0001 0000    32
_________   AND
 0000 0000     0
```

Das Ergebnis ist Null, der Computer wartet weiter.

Taste gedrückt:

```
 0010 0111    39
 0001 0000    32
_________   EOR
 0011 0111
 0001 0000    32
_________   AND
 0001 0000    32
```

Das Ergebnis ist ungleich Null, der Computer geht im Programm weiter.

Im Programmers Reference Guide steht es genau umgekehrt: Danach soll erst die AND und dann die EOR-Operation stattfinden. Lassen Sie uns diese Variante ebenfalls einmal durchspielen:

keine Taste:

```
 0011 0111    55
 0010 0000    32
_________   AND
 0010 0000
 0010 0000    32
_________   EOR
 0000 0000     0
```

Ergebnis also wieder Null. Der Computer wartet weiter.

Taste gedrückt:

```
 0010 0111    39
 0010 0000    32
_________   AND
 0010 0000
 0010 0000    32
_________   EOR
 0000 0000    0!
```

Auch wenn eine Taste gedrückt ist, würde bei dieser Funktionweise der Computer weiter warten bis zum jüngsten Gericht.

Wir sollten uns das merken: Es kommt bei mehreren logischen Verknüpfungen auch auf die Reihenfolge an.

Daß der WAIT-Befehl so selten benutzt wird, hängt zum einen sicherlich mit seiner relativ komplexen Handhabung zusammen, zum anderen aber auch damit, daß meist ein äußeres Ereignis eintreten muß, um den zu überprüfenden Bit-Wert zu verändern. Dafür kommen Kontrollregister in Frage, die für Ein-/Ausgabe-Operationen jeglicher Art eine Rolle spielen, einige VIC-II-Chip-Register und auch Speicherstellen des SID-Chip. Die beiden letzteren sind allerdings von der Firmware her ohnehin auf allerlei Ereignisse hin programmiert, so daß eine Behandlung mittels des WAIT-Befehls lediglich manchmal als Alternative gesehen werden kann. Bleiben also die CIA-Register, de-