MEMORY MAP mit Wandervorschlägen

Es steckt sehr viel im C 64. Wir werden Ihnen im Rahmen dieses Kurses die Bedeutung und Anwendung der Speicher und Register von Betriebssystem und Interpreter näherbringen.

Dieser Kurs ist eine zusammenhängende Wiedergabe einer Serie von Aufsätzen, die im 64'er, Ausgabe 11, November 1984 angefangen hat und sich in 18 Folgen bis Ausgabe 6, Juni 1986, hinzog.

Die vorliegende Fassung hat profitiert von vielen Zuschriften, die ich im Laufe des Kurses erhalten habe, mit Kritik, Hinweisen auf Fehler, Fragen und Verbesserungsvorschlägen. Für diese Mitarbeit der Leser möchte ich mich an dieser Stelle recht herzlich bedanken.

Ein Inhaltsverzeichnis der Adresse ist wenig sinnvoll. Ich habe allerdings die Adressen in einer Liste zusammengefaßt, geordnet nach ihren Funktionen, nicht nach ihrer Reihenfolge.

Dadurch soll Ihnen eine weitere Möglichkeit zum Nachschlagen geboten werden.

Auflisten kann ich lediglich die Texteinschübe und die Tabellen, zur Übersicht für eilige Sucher.

Hinweise und Tips über nützliche PEEK- und POKE-Adressen gehören zum Standard-Repertoire einer Computer-Zeitschrift. Ebenso häufig werden Leserfragen zu diesem Thema gestellt, obwohl mehrere Handbücher für die beiden Heimcomputer von Commodore bereits Speicherlisten (auf englisch »Memory Map«) enthalten.

Warum ich mich jetzt auch noch mit diesem Thema befassen will, hat zwei Gründe. Zum einen stört mich, daß ein Hinweis wie:
»...mit POKE 19,1 läßt sich das Fragezeichen bei INPUT-Befehlen unterdrücken...«
zwar richtig und auch anwendbar ist, aber halt nicht erklärt, was da eigentlich passiert und welche Folgen das für ein Programm haben kann. Zum anderen vermisse ich speziell in den Speicherlisten nähere, auch für den Anfänger verständliche und irgendwann einmal verwertbare Angaben.

Ich habe mir deshalb vorgenommen, Ihnen die Bedeutung

Texteinschub Nr. 1
Der USR-Befehl

Hand aufs Herz: Haben Sie USR schon einmal benutzt? Ohne Zweifel gehört dieser Befehl zu den seltenen. Ich will ihn daher hier kurz erläutern. USR hat dieselbe Funktion wie SYS, nämlich aus einem Basic-Programm direkt in ein Maschinenprogramm zu springen und dort solange weiterzufahren, bis mit dem Befehl RTS (entspricht dem Basic-Befehl RETURN) in das Basic-Programm zurückgesprungen wird. Die Sprungadresse in das Maschinenprogramm steht bei SYS gleich hinter dem Befehl.

Bei USR muß die Adresse zuerst in die Speicherzellen 1 und 2 (aha!!) gePOKEt werden.
Beispiel - Sprung auf 56524 ($DCCC):
mit SYS: `SYS 56524`
mit USR: `POKE 1,204:POKE 2,220:X=USR(Y)`

Kein Wunder, daß USR selten benutzt wird. Aber erstens ist er durch das POKEn der Low-/High-Byte-Darstellung aufgebläht und zweitens hat er auch wesentlich mehr Fähigkeiten als SYS.

Sein Argument, im obigen Beispiel also das »Y«, wird nämlich zuerst in den »Fließkomma-Akkumulator« FAC 1 (Floating Point Accumulator Nr. 1) gebracht, der sich in den Speicherzellen 97 bis 102 ($61 bis $66) befindet. Da wir ihn auf unserer Reise durch den Speicher noch treffen werden, brauche ich jetzt nicht näher darauf einzugehen. Wichtig ist lediglich, daß der Wert von »Y« dann vom angesprungenen Maschinenprogramm verarbeitet werden kann. Das Resultat kommt dann wieder in diesen FAC 1 und steht als Wert von X (siehe Beispiel oben) dem Basic-Programm zur Verfügung.

Mit USR kann man also Variable ins Maschinenprogramm zur Bearbeitung und zurück transferieren - und das ist der Unterschied zum SYS-Befehl. Ich möchte das an einem kleinen Beispiel demonstrieren. Statt allerdings ein Maschinenprogramm selbst zu schreiben, verwende ich beziehungsweise springe ich auf eine Routine des Betriebssystems, welches Werte des FAC 1 für mathematische Operationen verwendet.

Als mathematische Operation wähle ich das eingebaute Programm für INT, welches im VC 20 ab Speicherzelle 56524 ($DCCC) steht, im C 64 steht es ab 48332 ($BCCC). Dieses wollen wir verwenden:

In Zeile 10 definieren wir einen Wert für die Variable X, der in das Maschinenprogramm gebracht werden soll. Mit Zeile 20 bringen wir die Startadresse des Maschinenprogramms in die Speicherzellen 1 und 2.

Laut Kochrezept teilen wir die Adresse 56524 auf in ein Low-Byte = 204 und ein High-Byte = 220.

Der Befehl in Zeile 30 löst den ganzen USR-Vorgang aus, Zeile 40 gibt uns das Resultat.

```
10 Y=14.35
20 POKE 1,204:POKE 2,220
30 X=USR(Y)
40 PRINT X
```

Hinweis:
Entsprechend der anderen Adresse 48332 lautet die Zeile 20 beim C 64:

```
20 POKE 785,204:POKE 786,188
```

Nach RUN erhalten wir das Resultat 14, wie das Gesetz für INT es befiehlt. Natürlich hätten wir gleich PRINT INT (14.35) schreiben können, aber ich wollte ja nur demonstrieren. Der eigentliche Wert des USR-Befehls kommt hauptsächlich bei selbstgeschriebenen Maschinenprogrammen zum Zuge.

Sie können zur Übung im obigen Programm statt INT auch COS verwenden, indem Sie auf die Adresse 57935 ($E261) beziehungsweise beim C 64 auf 57938 ($E264) springen. Der Vergleich mit dem Basic-Befehl COS muß dasselbe Resultat ergeben.

Wer hat gemerkt, daß wir überhaupt nichts mit der Speicherzelle 0 gemacht haben, obwohl sie doch beim USR angeblich beteiligt ist?

Sie ist es wirklich, doch ohne unser Zutun. In diese Adresse wird beim Einschalten des Computers die Zahl 76 ($4C) geschrieben. Das ist der Code für den Maschinenbefehl »JMP«, der soviel bedeutet wie GOTO. Bei USR springt nämlich das Programm auf die Speicherzelle 0, findet dort den Sprungbefehl und in den nachfolgenden Zellen 1 und 2 die Sprungadresse - und führt den Sprung auch gleich aus.

und Anwendungen der PEEK- und POKEbaren Adressen, - sozusagen eine Wanderkarte mit Tourenvorschlägen und Sehenswürdigkeiten - in Form von Beispielen und Kochrezepten näherzubringen. Mir ist durchaus bewußt, daß das kein leichtes Unterfangen ist, da ich möglichst ohne Fach-Jargon auch für Nichttechniker verständlich bleiben möchte, und da die Zahl der zu behandelnden Adressen recht hoch ist. Ich werde also um Kompromisse wohl manchmal nicht herumkommen. Bevor wir anfangen, möchte ich noch einen kleinen »Arbeitsplan« machen.

■ Zur Methode:
Meine Erklärungen sind so aufgebaut, daß sie am besten vor dem Computer mit der Zeitschrift auf den Knien nachvollziehbar sind, also »Lies und Tipp«.

Längere Erklärungen oder Beispiele, die den Rahmen einer Adressenbeschreibung sprengen würden, oder die von generellem Interesse sind, werden in gesonderten Texteinschüben behandelt.

■ Zum Adressenbereich:
Prinzipiell sind natürlich alle RAM-Adressen (RAM = Lese- und Schreibspeicher) POKEbar und kämen daher in Betracht. Wir werden uns aber nur den Bereich von 0 bis 1023 vornehmen.

■ Zum Computer:
Der genannte Speicherbereich hat mit wenigen Ausnahmen für VC 20 und C 64 die gleiche Bedeutung. Ich werde daher beide Computer gleichzeitig behandeln und auf Unterschiede jeweils gezielt hinweisen.

■ Der erste Hinweis:
In Tabelle 1 sind die Unterschiede in groben Umrissen zusammengefaßt.

■ Zur Darstellung:
Die Kenntnis der Bedeutung dieser Speicherzellen kommt auch Programmen in Maschinensprache zugute. Ich gebe daher alle Adressen sowohl als Dezimal- als auch als Hexadezimalzahl (mit vorgestelltem »$«) an.

■ Zu den Adressen:
Wenn in die zur Diskussion stehenden Speicherzellen eine Adresse aus dem erlaubten Bereich 0 bis 65535 ($0 bis $FFFF) hineingeschrieben wird, geschieht das immer mit der Aufteilung in einen niederwertigen Teil (Low Byte) und einen höherwertigen Teil (High Byte). Das Rezept zur Umrechnung finden Sie im Texteinschub Nr. 2 »Die Low-/High-Byte-Darstellung«.

Adressen	Unterschied
0 - 2	sind verschieden
3 - 672	haben gleiche Funktionen
673 - 677	im VC 20 nicht benutzt
678 - 767	in beiden nicht benutzt
768 - 783	sind bei beiden gleich
784 - 787	im VC 20 nicht benutzt
788 - 819	haben gleiche Funktionen
820 - 1023	sind bei beiden gleich

Tabelle 1. Unterschiede zwischen VC 20 und C 64

Wozu brauchen das Betriebssystem und der Basic-Übersetzer RAM-Speicherzellen?

Auf den ersten Blick ist nicht verständlich, warum die Speicherzellen von 0 bis 1023 feste Bedeutung haben und für normale Programme nicht zur Verfügung stehen. Wenn sie schon, wie es heißt, vom Betriebssystem und dem Übersetzer-Programm verwendet werden, warum stehen sie dann nicht gleich im ROM-Speicher bei allen anderen Teilen dieser Systeme?

Ein Computer führt einen Programmschritt nach dem anderen aus, ganz stur, ohne eigene Entscheidungsfähigkeit, es sei denn, das Programm schreibt derartige Entscheidungen vor. Das Betriebssystem ist sozusagen im ROM eingefroren beziehungsweise festgeschrieben. Das würde aber bedeuten, daß der Computer keine Variationsmöglichkeiten hat, und daß alle Programme in gleicher Weise ablaufen. Aber das stimmt natürlich nicht! Alle Programme sind verschieden, sie belegen einen verschieden langen Speicherbereich und verarbeiten die unterschiedlichsten Variablen. Wir geben verschiedene Zeichen mit der Tastatur ein, der Computer wartet, bis eine Taste der Datasette gedrückt ist und so weiter.

Dafür braucht das Betriebssystem einen Speicherbereich, der variabel ist, in den es Zwischenwerte ablegen und später wieder auslesen kann.

Und das ist genau der Speicherbereich, der uns interessiert, nämlich von 0 bis 1023, womit wir wieder beim Thema wären.

Jetzt aber geht es los und zwar gleich in die vollen. Denn ausgerechnet die ersten drei Speicherzellen haben laut Tabelle bei beiden Computern eine verschiedene Bedeutung und zusätzlich gehören sie mit zu den kompliziertesten.

Adresse 0 bis 2 ($0 bis $2) beim VC 20:

Sprungbefehl und wählbare »Sprungadresse« des USR-Befehls.

Die drei Adressen werden bei der Abwicklung des Basic-Befehls USR verwendet.

Hinweis: Diesen drei Adressen des VC 20 entsprechen beim C 64 die Adressen 784 ($310) bis 786 ($312). Die folgenden Erklärungen gelten also entsprechend auch für den C 64. Mit dem Basic-Befehl USR wird ein Programm, das in Maschinensprache geschrieben ist, gestartet.

Wie das im einzelnen geht und welche Rolle dabei die Speicherzellen 0 bis 2 spielen, ist im Texteinschub Nr. 1 »Der USR-Befehl« näher beschrieben.

Adresse 0 ($0) beim C 64:

Datenrichtungsregister für Ein-/Ausgabe-Port des 6510-Mikroprozessors

Adresse 1 ($1) beim C 64:

Datenregister für Ein-/Ausgabe-Port des 6510-Mikroprozessors

Adresse 2 ($2) beim C 64:

unbenutzt

Im Gegensatz zum Mikroprozessor des VC 20 hat der des C 64 sechs Ein-/Ausgabe-Leitungen, die einzeln programmierbar sind und so eine direkte Verbindung zwischen dem Mikroprozessor und der Außenwelt herstellen. Warum nur sechs Leitungen und nicht wie üblich acht? Auf dem Chip selbst könnten acht Bit verkraftet werden, aber es stehen nur sechs Anschlußbeine zur Verfügung.

Um trotzdem flexibel zu bleiben, ist dieses Tor zum Prozessor - zutreffend auch »Port« genannt - in beiden Richtungen begehbar. Jede einzelne der sechs Leitungen kann vom Programmierer auf »Eingang« oder auf »Ausgang« geschaltet werden. Dazu dient das Datenrichtungsregister in der Speicherzelle 0.

Datenrichtungsregister in Zelle 0

Wenn zum Beispiel in das Bit 4 der Zelle 0 eine 0 hineingePOKEt wird, ist die Leitung Nummer 4 des Ports auf »Eingang« geschaltet. Es gilt für alle 6 Bit (Nummer 0 bis 5):
- Bit auf 0 = Eingang
- Bit auf 1 = Ausgang

Beim Einschalten schreibt das Betriebssystem in dieses Register die Dualzahl ..101111 (dezimal=47). Das heißt also, daß nur die Leitung Nummer 4 als Eingang verwendet wird, alle anderen aber als Ausgang. Warum das so ist, sehen wir gleich. Vorher will ich aber noch erwähnen, daß im C 64 von dieser Flexibilität des Mikroprozessor-Ports kein Gebrauch gemacht wird. Ich habe das ganze Betriebssystem durchgesehen, aber das einzige Mal, wo die Speicherzelle 0 angesprochen wird, ist eben bei der Einschaltroutine.

Das heißt aber nicht, daß Sie, lieber Hobby-Programmierer, darauf verzichten müssen. Ich kann mir vorstellen, daß besonders Ausgefuchste unter Ihnen durch POKEn eines anderen Bitmusters in die Speicherzelle 0 vielseitige Befehle erzeugen und einsetzen können.

Das wird besonders deutlich, wenn Sie jetzt sehen, mit welchen Teilen des Computers diese sechs Leitungen verbunden sind.

Datenregister in Speicherzelle 1

Mit diesem Register steuert der Mikroprozessor (und damit natürlich das Betriebssystem) die Auswahl von Speicherblöcken und den Betrieb mit dem Kassettenrecorder. Dem Programmierer steht diese Möglichkeit über POKEn auch zur Verfügung.

Bit 0
schaltet den Speicherbereich 40960 bis 49151 ($A000 bis $BFFF) zwischen dem Basic-Übersetzer (Interpreter) im ROM und freiem RAM um (Normalzustand=1).

Bit 1
schaltet den Speicherbereich 57344 bis 65535 ($E000 bis $FFFF) zwischen dem Betriebssystem (Kernel) im ROM und freiem RAM um (Normalzustand =1).

Bit 2
schaltet den Speicherbereich 53248 bis 57343 ($D000 bis $DFFF) zwischen Zeichen-ROM und Ein-/Ausgabe-ROM um (Normalzustand = 1).

Bit 3
sendet serielle Daten zum Kassettenrecorder (Normalzustand =0).
Bit 4
prüft, ob eine der Tasten des Recorders gedrückt ist, welche den Motor einschalten (Normalzustand =1).
Bit 5
schaltet den Motor des Recorders ein und aus (Normalzustand = 1).

Die RAM-ROM-Umschaltung

Sie wissen, daß Ihr C 64 deswegen so heißt, weil er 64 KByte Speicherplätze hat. Nur stimmt das nicht! Er hat nämlich 88 KByte und müßte eigentlich C 88 heißen.

Da mit den 16 Bit der High-/Low-Byte-Methode (siehe Texteinschub Nr. 2) nur 64 KByte adressierbar sind, müssen die restlichen 22KByte bei Bedarf eingeschoben werden - und das machen die oben erwähnten Bit 0 bis 2 des Datenregisters.

In Bild 1 sehen Sie die drei oben erwähnten Speicherblöcke, die sowohl mit RAM als auch mit ROM belegt sind, einer davon gleich doppelt. Ich habe ihnen folgende Namen gegeben:
- 40960 bis 49151 ($A000 bis $BFFF) = BLOCK A
- 53248 bis 57343 ($D000 bis $DFFF) = BLOCK D
- 57344 bis 65535 ($E000 bis $FFFF) = BLOCK E

Tabelle 2 gibt Ihnen die Übersicht über die gemeinsame Wirkung der Bit 0, 1 und 2 des Datenregisters auf den jeweiligen Inhalt der Speicherblöcke.

Der Vollständigkeit halber muß ich hier noch erwähnen, daß neben den drei ersten Bits der Speicherzelle 1 noch zwei weitere Signale die RAM/ROM-Umschaltung beeinflussen. Es sind das die Leitungen auf Pin 8 und 9 des Erweiterungssteckers (GAME und EXROM), welche durch Spiel- und Programmodule benutzt werden. Eine genaue Beschreibung der dadurch erzeugten sinnvollen Speicherkombinationen finden Sie in dem Buch »64 Intern« von Data Becker ab Seite 14. Zwei Anwendungsbeispiele dieser Umschaltung finden Sie im Texteinschub Nr. 3 »Manipuliertes Basic«.

Betrieb des Kassettenrecorders

Bit 3, 4 und 5 regeln, wie schon gesagt, den Betrieb des Kassettenrecorders.

Zu **Bit 3** ist oben schon alles Notwendige gesagt.

Bit 4 ist im Normalzustand auf 1, »normal« heißt hier, solange keine der Motor-Tasten der Datasette (PLAY, REWIND, FAST FORWARD) gedrückt ist. Zur Probe:

```
10 X=PEEK(1)
20 PRINT X
40 GOTO 10
```

Bit#	2	1	0	DEZ	HEX	BLOCK A	BLOCK D	BLOCK E
	1	1	1	55	$37	Basic	I/O	Kernal
	1	1	0	54	$36	RAM	I/O	Kernal
	1	0	1	53	$35	RAM	I/O	RAM
	1	0	0	52	$34	RAM	RAM	RAM
	0	1	1	51	$33	Basic	Zeichen	Kernal
	0	1	0	50	$32	RAM	Zeichen	Kernal
	0	0	1	49	$31	RAM	Zeichen	RAM
	0	0	0	48	$30	RAM	RAM	RAM

Dabei bedeuten
- Basic: Basic-Übersetzer (Interpreter)
- I/O: Ein-/Ausgabe-Register
- Zeichen: Zeichenspeicher
- Kernal: Betriebssystem
- RAM: frei verfügbarer Speicher

Tabelle 2. So sind die Bits 0, 1 und 2 des Datenregisters mit dem Inhalt der Blöcke A, D und E verknüpft.

Die schon erwähnte »Normalzahl« 55 (dual = 110111) läuft als Zahlenband solange, bis eine der besagten Tasten gedrückt wird. Dann läuft eine 7 (dual = 000111). Warum auch Bit 5 zu 0 wird, kommt später zur Sprache.

Mit einer kleinen Erweiterung der drei Zeilen können Sie in einem Programm den Status der Motor-Tasten abfragen. Ergänzen Sie:

```
30 IF X=7 THEN 50
50 PRINT »TASTE GEDRUECKT«
```

Um nur Bit 5 abzufragen, schreiben wir besser:

```
30 IF (X AND 16)=0 THEN 50
```

Diese Abfrage kann allerdings nicht unterscheiden, welche der drei Tasten der Datasette gedrückt worden ist. Außerdem funktioniert das alles nur, wenn - wie im »Normalfall« - das Bit 4 des Datenrichtungsregisters (Speicherzelle 0) auf 0 (Eingang) steht.

Bit 5 schaltet den Motor der Datasette ein und aus. Es bietet sich an, damit per Programm die Datasette zu schalten - wenn so etwas nützlich ist. Leider ist dieses Bit etwas schwieriger zu handhaben, da es in der Interrupt-Routine des Betriebssystems eine Rolle spielt.

Die Tasten der Datasette werden nämlich 60mal in der Sekunde abgefragt. Wenn keine Taste gedrückt ist, setzt das Betriebssystem sowohl das sogenannte »Interlock«-Register in Speicherzelle 192 auf 0 als auch Bit 5 der Zelle 1 auf 1, wodurch der Motor ausgeschaltet wird beziehungsweise

Texteinschub Nr. 2
Die Low-/High-Byte-Darstellung

Eine Speicherzelle der kleinen Commodore-Computer VC 20 und C64 hat eine Länge von 8 Bit = 1 Byte. Mit diesen 8 Bit können Zahlen von 0 bis 255 ($00FF) dargestellt werden. Zur Darstellung von Zahlen über 255 verwenden wir die Low-/High-Byte-Methode.

Wir hängen einfach zwei Speicherzellen zusammen, mit deren 16 Bit wir Zahlen bis maximal 65535 ($FFFF) darstellen können. Die maximale Zahl 65535 ist übrigens auch die höchste Adresse des gesamten Speichers - was natürlich kein Zufall ist.

Ich will Ihnen jetzt zeigen, wie eine Dezimalzahl auf zwei 8-Bit-Speicherzellen verteilt wird, und umgekehrt, wie aus 2 Byte eine Dezimalzahl gebildet wird.

Schauen Sie sich das folgende Beispiel an:

DEZIMAL	47491			
DUALZAHL	1011	1001	1000	0011
HEX $	B	9	8	3
HIGH-BYTE	185		-	
LOW-BYTE	-		131	

Wir gehen von der Dezimalzahl 47491 aus. Ihre duale Darstellung mit 16 Bit - 1011100110000011 - teilen wir einfach in der Mitte und erhalten damit zwei neue Dual-Zahlen mit je 8 Bit = 1 Byte. Das linke Byte nennen wir »High-Byte«, da es den höheren Teil der Gesamtzahl darstellt. Das rechte Byte heißt entsprechend »Low-Byte«.

Jedes der beiden Bytes kann für sich allein in einer Speicherzelle untergebracht werden, in der natürlich dann der dezimale Wert des Bytes steht.

In der Tabelle habe ich zur Vollständigkeit noch die hexadezimalen Werte eingefügt, die sehr schön zeigen, daß der Vorteil dieser Zahlendarstellung darin liegt, daß jede Einzelziffer der 4-Bit-Dualzahl entspricht, genau so wie jede Zweiergruppe dem Byte (sowohl in Dual-, als auch in Dezimaldarstellung) und die vierstellige Zahl der großen Dezimal- und Dualzahl entspricht.

Zur Umrechnung der Low-/High-Bytes empfehle ich folgende Kochrezepte:

Dezimal in Low-/High-Byte

47491:256=185(High-Byte), Rest 131 (Low-Byte)

Der Rest fällt bei der Division per Hand automatisch an. Mit dem (Taschen-)Rechner erhält man den Rest durch:

185*256-47491=-131

Low-/High-Byte in Dezimal

High-Byte * 256 + Low Byte = Dezimal
185 * 256 + 131 = 47491

Wichtige Regel: Die Mikroprozessoren von VC 20 und C64 verlangen, daß immer das Low-Byte vor dem High-Byte kommen muß. Die Zahl wird sozusagen von rechts nach links gelesen (131/185).

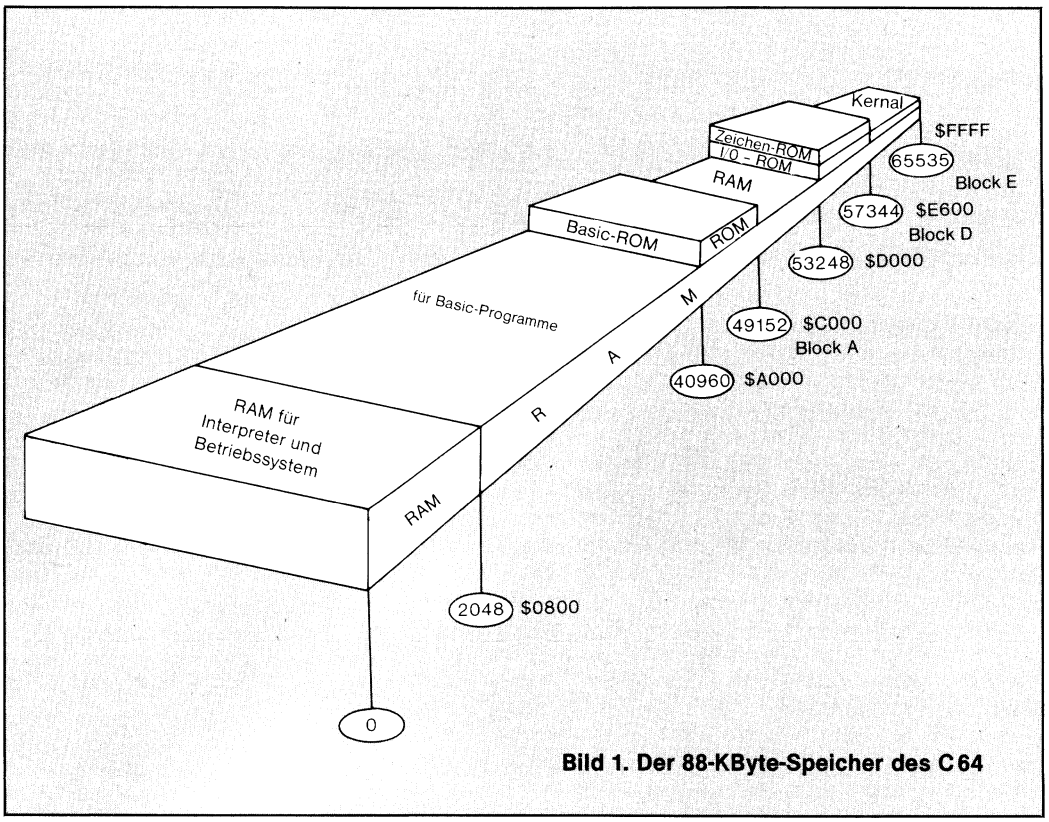


Bild 1. Der 88-KByte-Speicher des C64

bleibt. Da kann man nicht dagegen an. Wir haben nur eine Chance, wenn eine Taste bereits gedrückt ist und der Kassettenmotor schon läuft.

Dann nämlich können wir zuerst das Interlock-Register mit einem Wert größer als 0 lahmlegen:

```
POKE 192,255
```

Jetzt läßt sich der Motor der Datasette mit Bit 5 steuern:

```
POKE 1,39
```

beziehungsweise

```
POKE 1,PEEK(1) OR 32
```

schaltet den Motor aus,

```
POKE 1,7
```

beziehungsweise

```
POKE 1,PEEK(1) AND 31
```

schaltet den Motor ein.

Das Interlock-Register in Speicherzelle 192 werde ich später noch einmal erwähnen, da es seine Funktion auch beim VC 20 ausübt, allerdings mit anderen Ein-/Ausgangs-Ports. Das ist alles, was zur Speicherzelle 1 zu sagen ist.

Ab Speicherzelle 3 bis zur Speicherzelle 672 gelten alle

Texteinschub Nr. 3
Manipuliertes Basic

Wie Sie durch PRINT PEEK (1) selbst leicht feststellen, steht nach dem Einschalten des Computers im Register 1 die Zahl 55. In dualer Darstellung ist das 110111. Das entspricht dem in der ersten Zeile der Tabelle 2 dargestellten »Normalzustand« der einzelnen Bits.

Vergleichen Sie es bitte mit der Auflistung am Anfang der Beschreibung der Speicherzelle 1. Die in Tabelle 2 dargestellten Bits sind also die rechten drei Bits der Zelle 1.

Lassen wir die Bits 3, 4 und 5 unverändert, ergeben die acht Kombinationen der Tabelle 2 die Zahlen 55 bis 48. Durch den Befehl POKE 1,54 können wir nun den Basic-Übersetzer ausschalten und 8 KByte Speicher gewinnen. Nur nutzt uns das nicht viel, denn was tun - ohne Basic! Es gibt aber doch eine Anwendung. Zuvor will ich Ihnen aber noch beweisen, daß wir tatsächlich den Block A auf RAM umschalten. Der Trick besteht darin, den Basic-Übersetzer vom ROM in den darunter liegenden RAM umzuladen. Wenn er tatsächlich in RAM steht, müßten wir ihn durch POKEn verändern können zu einem Privat-Basic. Geben Sie direkt ein:

```
FOR J=40960 TO 49151: POKE J,
PEEK(J):
NEXT J
```

POKE J, PEEK (J) - das sieht dümmer aus als es ist. Die »Doppeldecker-Speicher« erlauben nämlich ein PEEKen nur aus dem ROM-Bereich. Ein hineinPOKEn dagegen geht nur in den RAM-Teil. Von dort aber kann er - wie gerade gesagt - nicht herausgelesen werden, es sei denn, wir schalten um!

Merken Sie was? Die Zeile oben liest also den Inhalt des Basic-ROMs und schreibt ihn in den RAM mit identischen Adressen. Die Ausführung der Zeile braucht einige Zeit. Wenn der Cursor wieder blinkt, schalten wir das RAM ein mit:

```
POKE 1,54
```

Wir merken natürlich noch keinen Unterschied, denn das RAM-Basic ist ja noch dasselbe.

Doch nun werden wir es verändern. In der Speicherzelle 41220 steht das »P« für den Befehl PRINT mit dem ASCII-Codewert 80. Dieses P ersetzen wir durch ein »G« (ASCII-Code = 71).

```
POKE 41220,71
```

Versuchen Sie bitte, mit dem (nicht durch »?« abgekürzten) PRINT-Befehl ein Zeichen auf den Bildschirm zu drucken. Es wird Ihnen nicht gelingen, denn der Befehl heißt jetzt:

```
GRINT "A"
```

was beweist, daß das Basic jetzt im RAM steht. Das Umdefinieren von Befehlen ist natürlich wenig sinnvoll. Aber wer die Maschinenprogramme des Basic kennt, kann sie auf diese Weise ändern, erweitern, einschränken, solange er sich auf in sich geschlossene Teile beschränkt.

Eine inzwischen oft zitierte Anwendung stammt von Jim Butterfield, den es begreiflicherweise stört, daß der Befehl ASC, welcher den ASCII-Code eines Strings erzeugt, bei einem Null-String das Programm mit ILLEGAL QUANTITY ERROR beendet. Versuchen Sie es:

```
PRINT ASC ("A") ergibt die Zahl 65.
PRINT ASC ("") hat die obige Fehlermeldung zur Folge.
```

Wenn Basic im RAM steht, können wir das ändern:

```
POKE 46991,5
```

Die Wiederholung des Befehls PRINT ASC ("") ergibt jetzt 0 - und, was das Wichtige ist, das Programm läuft weiter.

Durch zusätzliches Umladen des Speicherblocks E und anschließendes Umschalten mit POKE 1,53 ist auch das Betriebssystem veränderbar - ein weites Feld für fortgeschrittene Programmierer in Maschinensprache.

Die wohl wichtigste Anwendung der Umschaltmethode wird den Maschinen-Programmierern geboten, die dadurch eine kostenlose Speichererweiterung von 16 KByte erhalten. Bei gleichzeitiger Verwendung von Basic und Maschinenprogramm kann die Umschaltung besonders vorteilhaft eingesetzt werden. Das Umschaltprogramm muß dann aber ebenfalls in Maschinensprache geschrieben sein und darf nicht im Umschaltbereich liegen.

Das Umschalten von den Ein-/Ausgabe-Registern des Blocks D mit POKE 1,51 erlaubt, die Bitmuster der fest programmierten Zeichen aus dem Zeichen-ROM auszulesen, in einen freien RAM-Bereich zu bringen und dort dann nach eigenen Vorstellungen zu verändern.

Angaben sowohl für den C 64 als auch für den VC 20, zumindest was die Bedeutung der Zellen betrifft. Ihr Inhalt kann entsprechend der verschiedenen Adressen der Betriebssysteme voneinander abweichen. Wie üblich werde ich natürlich jeweils darauf aufmerksam machen.

Adresse 3 und 4 ($3 und $4)

Vektor auf die Routine zur Umwandlung einer Gleitkommazahl in eine ganze Zahl mit Vorzeichen

In diesen beiden Speicherzellen steht also ein Vektor. Was das ist, wird im Texteinschub Nr. 4 näher erläutert. Beim VC 20 deutet dieser Vektor auf die Adresse 53674 ($D1AA), beim C 64 auf 45482 ($B1AA). Sie können das mit

```
PRINT PEEK (3)+256*PEEK (4)
```

leicht nachprüfen. Ab diesen Adressen beginnt im Basic-Übersetzer (Interpreter) ein Programm, welches - natürlich in Maschinensprache - eine Gleitkommazahl in eine ganze Zahl umwandelt.

Diejenigen Leser, die mit Gleitkommazahlen nicht so vertraut sind, möchte ich auf den Texteinschub Nr. 5 verweisen. Er ist nur eine kleine Einführung. Eine detaillierte Beschreibung finden Sie im Assemblerkurs (Teil 8) von Heimo Ponnath (Ausgabe 4/85) beziehungsweise im 64er-Sonderheft 8/85, ab Seite 42.

Dieses Umwandlungsprogramm steht nicht nur den Maschinen, sondern auch den Basic-Programmierern zur Verfügung, allerdings nur über den USR-Befehl und da auch nur, wenn der »Floating Point Accumulator« #1 (FAC1) in den besagten Adressen 97 bis 102 mitbenutzt wird. Ich verschiebe daher alle weiteren Details auf unsere Ankunft bei diesen Speicherzellen.

Adresse 5 und 6 ($5 und $6)

Vektor auf die Routine zur Umwandlung einer ganzen Zahl in eine Gleitkommazahl

Dieses Programm ist die Umkehrung der oberen Routine. Sie beginnt beim VC 20 ab Speicherzelle 54161 ($D391), beim C 64 ab 45969 ($B391). Da hier prinzipiell dasselbe gilt wie oben, möchte ich nur kurz den Vorteil beleuchten, den derartige Vektoren haben. Eigentlich könnten wir direkt auf die im Vektor enthaltenen Adressen springen - wenn wir sie kennen.

Ein Sprung auf die Adresse des Vektors erlaubt uns jedoch immer die völlige Ignoranz seines Inhalts - und Commodore erlaubt die Änderung der Adressen im Basic-Übersetzer, wie es ja beim C 64 gegenüber dem VC 20 auch gemacht worden ist, ohne daß vorhandene Programme umgeschrieben werden müssen.

Adresse 7 ($7)

Suchzeichen zur Prüfung von Texteingaben in Basic

Diese Speicherzelle wird viel von denjenigen Basic-Routinen als Zwischenspeicher benutzt, die den direkt eingegebenen Text absuchen, um Steuerzeichen (Gänsefüße, Kommata, Doppelpunkte und die Zeilenbeendigung durch die RETURN-Taste) rechtzeitig zu erkennen. Normalerweise wird in der Zelle 7 der ASCII-Wert dieser Zeichen abgelegt. Die Speicherzelle 7 wird aber auch von anderen Basic-Routinen benutzt. Sie ist daher für den Programmierer praktisch nicht zu verwerten.

Adresse 8 ($8)

Suchzeichen speziell für Befehlsende und Gänsefüße

Wie Speicherzelle 7 dient auch die Zelle 8 als Zwischenspeicher für Basic-Texteingabe und zwar während der Umwandlung von Basic-Befehlen in den vom Computer verwendeten Befehlscode (Tokens). Die Speicherzelle 8 ist in Basic nicht verwertbar.

Adresse 9 ($9)

Spaltenposition des Cursors vor dem letzten TAB- oder SPC-Befehl

Speicherzelle 9 wird von den Basic-Befehlen TAB und SPC verwendet. Vor ihrer Ausführung wird die Nummer der Spalte, in der sich der Cursor befindet, aus der Speicherzelle 211 ($D3) nach 9 gebracht, von wo sie geholt wird, um die Position des Cursors nach der Ausführung von TAB und SPC auszurechnen.

Diese komplizierte Erklärung können wir durch Ausprobieren deutlicher machen. Dazu PRINTen wir 16mal den Buchstaben X hintereinander (Semikolon !), allerdings mit SPC (2) jeweils um 2 Spalten versetzt.

```
10 FOR I=0 TO 15
20 PRINT SPC (2) "X";
30 PRINT PEEK (9);
40 NEXT I
```

Nach jedem X wird durch Zeile 30 die »alte« Cursor-Spaltenposition ausgedruckt und zwar in derselben Zeile, ausgelöst durch das Semikolon. Dadurch erhöht sich laufend die in Speicherzelle 9 stehende Positionsangabe des Cursors. Wir erhalten folgenden Ausdruck:

```
. . X . O . . . X . 6 . . . X . 12 . . . X
. 19 . . . X . 26 . . . X . 33 . . . X .
40 . . . X . 47 . . . X . 54 . . . X . 6
1 . . . X . 68 . . . X . 75 . . . X . 82
. . . X . 1 . . . X . 7 . . . X . 13 . . .
```

Sie können die Positionsnummer nachrechnen. Berücksichtigen Sie aber dabei, daß bei PRINT vor und nach jeder Zahl eine Stelle frei bleibt, die erste für das Vorzeichen, die zweite wegen des Abstandes.

Wichtig ist außerdem, daß die maximal mögliche Spaltenzahl nicht die Bildschirmspaltenzahl, sondern die »logische« Spaltenzahl ist, also 88 beim VC 20 und 80 beim C 64.

Wir können die Cursorposition in Adresse 9 auch abfragen und ein Programm damit steuern. Fügen Sie einfach in das obige Programm die folgende Zeile 35 ein:

```
35 IF PEEK (9)=33 THEN PRINT
"END": END
```

Sobald Position 33 erreicht ist, bleibt das Programm stehen.

Adresse 10 ($A)

Flagge für LOAD oder VERIFY

In Zelle 10 steht eine 0, wenn geladen wird und eine 1 bei einem VERIFY. Warum das so ist, will ich kurz erläutern:

Die Basic-Routinen für LOAD beziehungsweise für VERIFY sind völlig identisch. Was das Betriebssystem hinterher daraus machen muß, ist natürlich unterschiedlich. Das Basic erspart sich eine doppelte Routine, zeigt aber mit der Flagge in Speicherzelle 10 den Unterschied an.

Erwähnenswert ist noch, daß das Betriebssystem in einer Art Nationalismus seine eigene Flagge aufzieht: Den Unterschied zwischen LOAD und VERIFY speichert es seinerseits in Zelle 147 ($93) ab. Soweit ich es sehen kann, sind Inhalt und Bedeutung beider Speicherzellen völlig identisch.

Ich habe für Sie zwar kein Kochrezept zur Anwendung der LOAD-VERIFY-Flagge in einem Programm vorrätig, möchte Sie aber trotzdem ein bißchen zum Spielen anregen. Um meine Erklärung nachzuvollziehen, tippen Sie bitte direkt LOAD ein. Den Ladevorgang brechen Sie mit der STOP-Taste ab und fragen dann den Inhalt der Zelle 10 ab mit

```
PRINT PEEK (10)
```

Wir erhalten eine 0.

Wiederholen Sie bitte diesen Vorgang, aber mit VERIFY. Wir erhalten jetzt eine 1 - Quod erat demonstrandum.

Wir können auch in die Zelle 10 hineinPOKEn. Die »Wachablösung« zwischen Basic und Betriebssystem unter Hissen der Flagge in Zelle 10 findet beim VC 20 in der Speicherzelle 57705, beim C 64 in 57708 statt. Bevor wir diese Maschinenroutine mit SYS 57705 (SYS 57708) starten, geben wir mit dem Inhalt der Speicherzelle 10 an, ob es ein LOAD oder ein VERIFY sein soll.

Legen Sie ein Band mit Programm in die Datasette. Um ein LOAD zu erzeugen, geben wir direkt ein:

```
POKE 10,0:SYS 57705
(POKE 10,0:SYS 57708)
```

Entsprechend der Anweisung auf dem Bildschirm drücken Sie PLAY, und das Auffinden des ersten Programms wird mit LOAD gemeldet. Machen Sie das Ganze noch einmal, diesmal aber POKEn Sie bitte eine 1 in die Zelle 10. Jetzt meldet das Betriebssystem das Auffinden des Programms mit VERIFY.

Wie gesagt, vielleicht fällt Ihnen eine Anwendung dafür ein.

Adresse 11 ($B)

Flagge für den Eingabepuffer/Anzahl der Dimensionen von Zahlenfeldern (Arrays)

Alle Buchstaben und Zeichen, die mit der Tastatur direkt eingetippt werden, kommen in einen Eingabe-Pufferspeicher. Er beginnt ab Speicherzelle 512 ($200). Sobald die RETURN-Taste gedrückt wird, wandelt eine Routine des Basic-Übersetzers den Text in Codezahlen (Tokens) um. Diese Routine und eine andere, welche die Zeilen eines Programms aneinanderhängt, verwenden die Zelle 11 als Zwischenspeicher.

Sobald die Textumwandlung beendet ist, steht in Zelle 11 eine Zahl, die die Länge der Token-Zeile angibt.

Die Zelle 11 wird außerdem

noch von den Basic-Routinen benutzt, die ein Feld (Array) aufbauen oder ein bestimmtes Element in einem Array suchen. Was ein Feld oder Array ist, finden Sie in den Commodore-Handbüchern gut beschrieben. Außerdem gehe ich bei der Behandlung der Speicherzellen 47 bis 50 näher darauf ein.

Diese Routinen also verwenden die Speicherzelle 11, um die Anzahl der verlangten DIMensionen und den für ein neu aufgebautes Feld nötigen Speicherbedarf zu berechnen.

Adresse 12 ($C)

Flagge für Basic-Routinen, die ein Feld (Array) suchen beziehungsweise aufbauen

Diese Speicherzelle wird von den Basic-Routinen als Zwischenspeicher benutzt, die feststellen, ob eine Variable ein Feld (Array) ist, ob das Feld bereits DIMensioniert worden ist, oder ob ein neues Feld die unDIMensionierte Zahl von 11 Elementen hat.

Adresse 13 ($D)

Flagge zur Bestimmung des Datentyps (Zeichenkette/ String oder Zahl)

Diese Flagge zeigt den Routinen des Basic-Übersetzers an, ob es sich bei den zur Verarbeitung anstehenden Daten um einen String oder um Zahlenwerte handelt. Zeigt die Flagge 255 ($FF), ist es ein String. Bei 0 handelt es sich um Zahlen. Diese Bestimmung erfolgt jedesmal, wenn eine Variable definiert oder gesucht wird. Diese Flagge kann leider nicht durch ein Basic-Programm abgefragt werden.

Adresse 14 ($E)

Flagge zur Bestimmung des Zahlentyps (Ganze Zahl oder Gleitkommazahl)

Sobald durch die Flagge in der vorherigen Zelle 13 eine Zahl signalisiert wird, steht hier die Zahl 128 ($80), wenn es sich um eine ganze Zahl handelt, während eine 0 die Zahl als Gleitkommazahl identifiziert.

Damit wollen wir ein bißchen experimentieren. Zeile 10 definiert eine Gleitkommazahl, Zeile 20 druckt sie und die Flagge aus Zelle 14 aus.

```
10 A=13.41
20 PRINT A,PEEK (14)
```

Wir erhalten die Zahl 13.41 und als Flagge eine 0.

```
30 B=INT (A)
40 PRINT B,PEEK (14)
```

INT bildet die ganze Zahl von 13.41. Also müßte die Flagge in Zelle 14 auf 128 stehen. Weit gefehlt! Da intern auch die 13 als Gleitkommazahl berechnet wird, erhalten wir immer noch eine 0.

```
50 B%=A
60 PRINT B%,PEEK (14)
```

Erst die Definition der Variablen B als ganze Zahl (mit %) ergibt die Flagge 128.

```
70 D=16*B%
80 PRINT D,PEEK (14)
```

Die Multiplikation einer ganzen Zahl mit der Ganzzahl-Variablen B% fällt in dieselbe Kategorie wie Zeile 30 oben, da die Verarbeitung als Gleitkommazahl erfolgt. Also erhalten wir zu Recht eine 0. Erst wenn D als ganze Zahl (Zeile 90) ausgewiesen wird, steht die Flagge wieder auf 128:

```
90 D%=16*B%
100 PRINT D%, PEEK (14)
```

Adresse 15 ($F)

Flagge bei LIST, Garbage Collection und Textumwandlung

Die Routine des LIST-Befehls muß unterscheiden zwischen Basic-Befehlen und normalem Text. Wenn eine Zeichenkette durch ein »Gänsefüßchen« identifiziert worden ist, wird die Flagge gesetzt, und der Text wird ausgedruckt.

Unter »Garbage Collection« (Müllabfuhr) wird die Routine des Betriebssystems verstanden, welche zu bestimmten Anlässen im Variablenspeicher alle nicht mehr benötigten Strings entfernt, um Platz zu schaffen. Dabei wird eine Flagge in Zelle 15 gesetzt, die anzeigt, daß eine Müllabfuhr bereits stattgefunden hat. Wenn bei der Speicherung eines neuen Strings zu wenig Speicherplatz vorhanden ist, wird bei der Flagge nachgesehen, ob gerade vorher schon durch die Müllabfuhr (Garbage Collection) der Speicher entrümpelt worden ist. Falls das der Fall ist, wird OUT OF MEMORY angezeigt, falls nicht, wird eine Müllabfuhr durchgeführt.

Schließlich wird Zelle 15 auch bei der Umwandlung von Basic-Befehlen in internen Codezahlen (Tokens) eingesetzt.

Adresse 16 ($10)

Flagge zur Anzeige eines Variablen-Feldes oder einer selbstdefinierten Funktion

Im Basic-Übersetzer gibt es eine Routine, die den Speicher absucht, ob es eine Variable mit bestimmten Namen bereits gibt. Wenn diese mit einer Klammer beginnt, wird die Flagge in Zelle 16 gesetzt, um anzuzeigen, daß es sich um eine Array-Variable oder um eine mit DEF FN selbstdefinierte Funktion handelt.

Adresse 17 ($11)

Flagge für INPUT, GET oder READ

Die Basic-Routinen für INPUT, GET und READ sind zum großen Teil identisch. Um Speicherplatz zu sparen, verwendet der Basic-Übersetzer die identischen Teile nur einmal. Um in die nichtidentischen Teile verzweigen zu können, wird in Zelle 17 angezeigt, um welchen der drei Befehle es sich gerade handelt. Die Flagge steht auf 0 für INPUT, auf 64 ($40) für GET und auf 152 ($98) für READ.

Mit dem folgenden kleinen Programm können wir das leicht nachprüfen.

Texteinschub Nr. 4
Zeiger, Vektoren und Flaggen

Zeiger und Vektoren sind Zahlenwerte, die jeweils in zwei benachbarten Speicherzellen stehen und in der Low-/High-Byte-Darstellung eine Adresse bilden.

Wir sprechen von einem »Zeiger«, wenn diese Adresse den Beginn von gespeicherten Daten angibt.

Ein »Vektor« zeigt ebenfalls auf eine Anfangsadresse, allerdings auf die eines Maschinenprogramms. Diese Unterscheidung wird leider nicht immer ganz eindeutig angewendet.

Eine »Flagge« besteht aus einem Zahlenwert in einer Speicherzelle, die von einem Programm dort abgelegt wird, um sich das Resultat einer Operation zu merken, beziehungsweise um es für eine spätere Verwendung bereitzuhalten.

Texteinschub Nr. 5
Die Zahlendarstellung bei den Commodore-Computern

Gleitkomma-Zahlen

Für diejenigen Leser, die das Thema der Zahlendarstellung in den Commodore-Handbüchern großzügig übersprungen haben, stelle ich es hier noch einmal vor.

Sie kennen die gängigen vier Zahlentypen:

- ganze Zahlen: 15, 21, 244
- Brüche; $\frac{2}{3}$, $\frac{26}{8}$, $\frac{15}{14}$
- negative Zahlen: -15, -255
- positive Zahlen: 10, 5, 123

Ganze Zahlen bereiten uns und dem Computer keine Probleme.

Bei Brüchen sieht es schon anders aus. Erinnern Sie sich an die Bruchrechnungsstunden in der Schule? Wieviel ist $51\frac{1}{52} + \frac{3}{4}$!!!

Ohne lange zu überlegen, rechnen wir natürlich um, $51\frac{1}{52}$ = 0,9807692 und $\frac{3}{4}$ = 0,75; addiert ist das Resultat 1,7307692 - und schon sind Sie mitten in den Gleitkomma-Zahlen.

Bei obigem Beispiel gleitet allerdings noch nichts. Bei sehr großen oder aber auch sehr kleinen Bruch-Zahlen reicht uns - und einem Computer - nicht der Platz, um sie darzustellen. Die Zahl 0,000000000000000123 sprengt jeden normalen Rahmen.

Daher schreiben wir sie anders. Wir lassen das Komma nach rechts gleiten, bis es die erste Ziffer, die von 0 verschieden ist, findet und für jede Null, die es passiert, multiplizieren wir die Zahl mit 10.

Die Zahl oben sieht dann so aus:
0,123 x 10 hoch 15 (eine 1 mit 15 Nullen).

Die Grundzahl vorn heißt »Mantisse«, die 10 mit Hochzahl heißt »Exponent«.

Alle Commodore-Computer verarbeiten intern alle Zahlen in dieser Darstellung, also als Gleitkommazahl.

```
10 DATA 3
20 READ A
30 PRINT PEEK (17)
40 INPUT B
50 PRINT PEEK (17)
60 GET C$:IF C$= ""THEN 60
70 PRINT PEEK (17)
```

Zeile 10 und 20, 40 sowie 60 sind Anwendungen der drei zur Debatte stehenden Basic-Befehle. Nach der Durchführung jedes Befehls wird in den Zeilen 30, 50 und 70 die jeweilige Flagge ausgelesen.

Nach RUN erhalten wir als Resultat der Zeile 20 die Zahl 152, als Resultat von Zeile 30 die INPUT-Aufforderung mit Fragezeichen. Geben Sie irgendeine Zahl und RETURN ein. Wir erhalten so die 0. Die GET-Schleife in Zeile 40 wartet auf einen Tastendruck, dann erhalten wir 64.

Adresse 18 ($12)

1. Flagge für Vorzeichen bei SIN, COS und TAN
2. Flagge bei Vergleich

Zuerst kommt das Vorzeichen der trigonometrischen Funktionen an die Reihe.

Die Routinen des Basic-Übersetzers (Interpreter), welche die drei trigonometrischen Funktionen SIN, COS und TAN berechnen, verwenden die Speicherzelle 18 zur Bestimmung des Vorzeichens.

Zur Erinnerung: Die trigonometrischen Funktionen haben in den vier »Quadranten« des Kreises (0-90, 90-180, 180-270, 270-360 Grad) nicht unbedingt dieselben Vorzeichen. Die Vorzeichen ändern sich allerdings nur an den Grenzen der Quadranten, wie in Bild 2 zu sehen ist. Die Flagge in Zelle 18 gibt das Vorzeichen nicht direkt an, sondern auf Umwegen. Die Darstellung ist in der folgenden Tabelle zusammengefaßt.

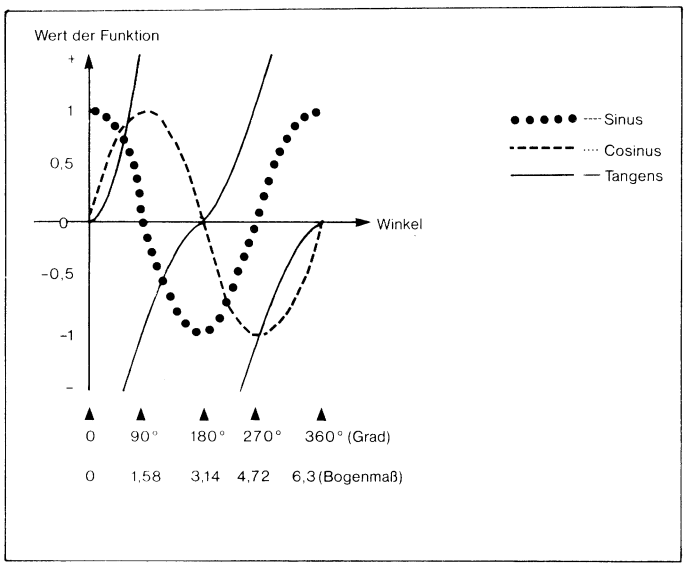


Bild 2. Trigonometrische Funktionen SIN, COS, TAN

WINKEL	0-90	90-180	180-270	270-360
SIN	gleich	Wechsel	Wechsel	gleich
COS	255	255	0	0
TAN	0	255	255	0

Dabei bedeutet »gleich«: 0-0-0-0 oder 255-255-255
»Wechsel«: 0-255-0-255

Da die Erklärung mit »gleich« beziehungsweise »Wechsel« nicht gerade einleuchtend ist, schlage ich vor, daß Sie sich das Ganze mit dem folgenden kleinen Programm selbst anschauen, welches für viele Werte des Winkels im Bogenmaß - und in kleinen Schritten - den Wert der Flagge, daneben den Winkel I und den Wert der Funktion mit Vorzeichen ausdruckt.

```
10 FOR I=0 TO 10 STEP 0.01
20 PRINT PEEK(18);INT
(I*100)/100;SIN(I):NEXT
```

Diese etwas umständliche Art, den Wert von I auszudrucken, vermeidet Rundungsfehler und begrenzt den Ausdruck auf zwei Dezimalstellen. Wenn Sie die Winkelwerte von I in Graden ausgedruckt haben wollen, können Sie eine andere Zeile 20 verwenden, welche die Umrechnungsformel vom Bogenmaß in Grade verwendet:
Winkel in Grad = Winkel im Bogenmaß * $180/\pi$

```
20 Print PEEK(18);INT
(I*180/π);SIN(I):NEXT
```

Statt SIN können Sie genauso gut COS und TAN einsetzen.

In Bild 2 sind nicht nur die Kurven und die Bereiche der Vorzeichen, sondern auch die Winkelbereiche sowohl im Bogenmaß als auch in Graden dargestellt.

Die Speicherzelle 18 wird auch noch von anderen Routinen des Basic-Interpreters beansprucht und zwar von allen, die einen Vergleich wie <, >, >= und so weiter durchführen. Entsprechend der Art des Vergleichs steht dann in der Zelle 18 eine Ziffer von 0 bis 6.

Das folgende Programm macht das deutlich.

```
10 A=2
20 FOR I=1 TO 3
30 IF I= A THEN PRINT I;
PEEK(18);"="
40 IF I< >A THEN PRINT I;
PEEK(18);" > < "
50 IF I> A THEN PRINT I;
PEEK(18);" > "
60 IF I< A THEN PRINT I;
PEEK(18);" < "
70 IF I>=A THEN PRINT I;
PEEK(18);" > "="
80 IF I<=A THEN PRINT I;
PEEK(18);" < "="
90 IF I<A OR I=A THEN
PRINT I;PEEK(18);" < OR ="
100 NEXT I
```

Kurz zur Erklärung dieser Zeilen: In der FOR..NEXT-Schleife wird die Variable I mit der Konstanten A=2 verglichen. In den Zeilen 30 bis 90 werden alle möglichen Vergleichsoperatoren durchgeprüft. Jeder der zutrifft, druckt den Wert von I, den Wert der dann in Zelle 18 stehenden Flagge und schließlich den Vergleichsoperator aus. Aus dem Resultat dieses Programms läßt sich folgende Tabelle zusammenstellen:

Vergleich	Flagge in 18
< OR =	0
> OR =	0
>	1
=	2
>=	3
<	4
< >	5
<=	6

Sie sehen, die Flagge für die kombinierten Vergleichsoperatoren entspricht der Summe ihrer Einzelwerte. Nur die Verknüpfung über OR nicht, denn die ergibt 0.

Adresse 19 ($13)

Flagge zur Kennzeichnung des laufenden Ein-/Ausgabegerätes

Immer dann, wenn von Basic Daten ein- oder ausgegeben werden, schaut die entsprechende Routine des Übersetzers in Zelle 19 nach, um welches Peripheriegerät es sich handelt. Zur Debatte stehen Tastatur, Datasette, RS232-User-Port, Bildschirm, Drucker und Floppy-Laufwerk.

Die Flagge ihrerseits ist ausschlaggebend für die feinen Unterschiede, wie zum Beispiel das Fragezeichen, bei Eingabe von der Tastatur (INPUT) oder die Anweisung »Press Play on Tape« bei Eingabe von der Datasette.

Beim Einschalten des Rechners setzt die Initialisierungsroutine des Betriebssystems, die beim VC 20 ab Adresse 58276 ($E3A4), beim C 64 ab 58303 ($E3BF) beginnt, die Flagge in Zelle 19 auf 0. Die Null bedeutet Eingabe über Tastatur und Ausgabe über Bildschirm.

Wenn Sie einen Disassembler haben, drucken Sie doch einmal das Assemblerlisting aus. Sie werden in Adresse 58324/58325 ($E3D4/E3D5), beim C 64 in 58354/58355 ($E3F2/E3F3) den Befehl finden, der eine Null nach Zelle 19 ($13) bringt.

Immer dann, wenn ein Programm nicht Tastatur und Bildschirm, sondern eines der oben genannten anderen Peripheriegeräte anspricht (indem mit OPEN.... eine Datei = Logical File eröffnet wird), wird in Zelle 19 die Nummer der gerade bearbeiteten Datei eingetragen, mit den bereits beschriebenen Konsequenzen.

Ich will hier nicht weiter darauf eingehen, da wir den Inhalt von Zelle 19 selbst nicht auslesen können. Er wird nämlich immer gleich wieder auf Null gesetzt.

Wir können ihn aber durch POKE verändern. Durch POKE 19,1 gaukeln wir dem Rechner vor, daß Ein- und Ausgabe über »externe« Geräte läuft, selbst wenn nur die Tastatur und der Bildschirm betrieben werden.

Wenn zum Beispiel der Rechner der Meinung ist, daß ein INPUT von der Datasette kommt, druckt er kein Fragezeichen aus; auch kein EXTRA IGNORED als Fehlermeldung bei zu zahlreicher Eingabe und das alleinige Drücken der RETURN-Taste ignoriert er auch, im Gegensatz zum »normalen« INPUT.
Probieren Sie es aus:

```
10 INPUT "TEST";A$
20 PRINT A$
```

In diesem Normalfall erscheint nach RUN darunter die Aufforderung TEST?

Eine Eingabe, zum Beispiel XX, erscheint mit einem Abstand daneben, und nach RETURN wird XX an den Anfang der nächsten Zeile gedruckt. Alle falschen Eingaben werden mit den üblichen Fehlermeldungen quittiert.

Jetzt fügen wir ein:

```
5 POKE 19,1
```

Nach RUN erscheint wieder die Aufforderung TEST, aber ohne Fragezeichen. Die Eingabe XX wird ohne Abstand daneben gesetzt und nach RETURN mit einem Abstand in derselben Zeile weitergeschrieben.

Das Drücken der RETURN-Taste setzt den Cursor nicht wie üblich in die nächste Zeile, sondern schiebt ihn in derselben Zeile weiter.

Diesen zusätzlichen Effekt muß man beachten, da er sehr störend für den Verlauf eines Programms sein kann.

Man kann ihn natürlich auch nutzbringend einsetzen, hat er doch die Eigenschaft eines automatischen »Cursor UP«. Eine pfiffige Anwendung dieser Art wurde von Brad Templeton für den PET erfunden und ist von Jim Butterfield für eine MERGE-Routine mit dem Namen »Magic Merge« veröffentlicht worden.

Da diese Routine aber primär auf der Eigenschaft der Speicherzelle 153 basiert, werde ich sie dann erläutern, sobald wir bei der Zelle 153 angelangt sind.

Zurück zur Flagge in Zelle 19.

Umgekehrt können wir POKE 19,0 leider nicht nutzen, da die betroffenen Befehle GET, GET#, INPUT, INPUT# und PRINT# die Flagge sofort auf den richtigen Wert setzen. Nur PRINT und LIST tun das nicht, wie wir bei dem PRINT-Befehl oben ja gesehen haben.

Adresse 20 und 21 ($14 und $15)

Zeilennummer für LIST, GOTO, GOSUB und ON, Zeiger der Adresse bei PEEK, POKE, SYS und WAIT

In diesen Speicherzellen wird die Zeilennummer der Sprungbefehle GOTO, ON..GOTO und GOSUB sowie die Zeilenangabe beim LIST-Befehl gespeichert. Da die Werte bis maximal 65535 gehen können, braucht der Computer 2 Byte zur High-/Low-Byte-Darstellung.

Die GOTO-Routine (im VC 20 ab 51360 = $C8A0, im C 64 ab 43168 = $A8A0) vergleicht die Zahl in 20 und 21 mit der laufenden Zeilenzahl. Wenn sie kleiner ist, wird ab der ersten Zeile des Programms gesucht. Ist sie aber größer, dann beginnt die Suche ab der laufenden Zeilenzahl. Die Suche geht solange, bis die in 20 und 21 angegebene Zeilenzahl gefunden ist. Dann fährt das Programm mit dieser Zeile fort.

LIST speichert in 20 und 21 die höchste auszulistende Zeilennummer ab, falls keine Angabe beim LISTen gegeben worden ist, den Wert 65535 ($FFFF).

Die Befehle PEEK, POKE, SYS und WAIT verwenden diese Speicherzellen zur Angabe der Adressen, die dem Befehl immer folgen müssen.

Leider können wir die Speicherzellen 20 und 21 mit Basic-Programmen nicht bearbeiten; ihr Inhalt wird immer gleich auf 20 zurückgesetzt.

Adresse 22 ($16)

Zeiger auf den nächsten freien Speicherplatz im »Temporary String Descriptor Stack«

Dieser Zeiger bezieht sich in seiner Wirkung auf die übernächsten Speicherzellen 25 bis 33 ($19 bis $21).

Diese werden als Stapelspeicher (Stack) für Angaben über vorläufige Zeichenketten - auf englisch »Temporary String Descriptor« - verwendet.

Die Speicherzelle 22 ($16) ihrerseits enthält einen Zeiger auf den jeweils nächsten verfügbaren Platz in diesem Speicher ab Zelle 25. Da er eine Kapazität von 3 * 3 Byte hat, zeigt der Zeiger auf die Zelle 25 ($19), wenn er leer ist. Bei einem Eintrag zeigt er auf 28 ($1C), bei zwei Einträgen auf 31 ($1F) und schließlich auf 34 ($22), wenn der Speicher voll ist.

Eine Zeichenkette ist dann »vorläufig«, wenn sie noch nicht einer Stringvariablen zugeordnet worden ist, zum Beispiel »Mahlzeit« in dem Basic-Befehl

```
PRINT "MAHLZEIT".
```

Beim Einschalten setzt das Betriebssystem mit der Einschaltroutine ab Adresse 58303 ($E3BF) im C 64, beim VC 20 ab 58276 ($E3A4) den Zeiger auf 25. Die Stringverwaltungsroutine ab 46215 ($B487) im C 64 beziehungsweise ab 54407 ($D487) im VC 20 fragt bei String-Eingaben die Flagge ab. Nach jeder Eintragung in den Speicher ab Zelle 25 wird der Zeiger um 3 weitergesetzt.

Sie können die Leerflagge 25 mit

```
PRINT PEEK (22)
```

leicht nachprüfen.

Die anderen Eintragungen können nicht nachgeprüft werden, weil sie sofort auf 25 zurückgesetzt werden.

Wir können sie aber durch POKE beeinflussen; ob das sinnvoll ist, ist eine andere Frage.

```
10 POKE 22,34
20 PRINT "MAHLZEIT"
```

Die Zahl 34 in Zelle 22 sagt dem Programm, daß der Speicher ab Zelle 25 voll ist. Wir bekommen statt der MAHLZEIT eine Fehlermeldung serviert.

Mit einem POKE-Befehl, der als Argument die für den vorgesehenen Zweck ungültige Zahl 35 verwendet:

```
POKE 22,35
```

erreichen wir allerdings zwei interessante »Dreckeffekte«. Zum einen unterdrückt der Befehl die Ausgabe des READY, zum anderen aber bewirkt er, daß bei LIST ein Listing ohne Zeilennummern ausgedruckt wird, sowohl auf dem Bildschirm als auch mit dem Drucker.

Das billigste editierfähige Textverarbeitungssystem

Die Idee dazu habe ich von Mike Apsey's Hinweis in »Commodore User« Juli 1984. Mit Zeilennummern versehen, läßt sich jeder beliebige Text schreiben, verbessern, verschieben, abspeichern, aber nicht RUNen!!

Der POKE-Befehl von oben (POKE 22,35) gefolgt von einem CMD und LIST, druckt dann alles brav als reinen Text aus. Die maximale Zeilenlänge entspricht der Zeilenlänge des jeweiligen Computers.

Probieren Sie es aus:

```
10 DER COMPUTER BIETET
IN DER
20 DATENFERNÜBER-
TRAGUNG
30 UNGEAHNTE
MÖGLICHKEITEN.
40 ABER DIE GEFAHR
50 USW. USW.
60:
```

Jede Zeile wird mit der RETURN-Taste abgeschlossen. Damit auch alles gedruckt wird, muß - zumindest bei meinem Drucker (1526) - eine »Leerzeile« folgen (Zeile 60). Mit

```
POKE 22,35:OPEN 1,4:CMD
1:LIST
```

wird der Text ohne Zeilennummern ausgedruckt. Sie können ihn vorher nach Belieben verändern.

Wie gesagt, nur nicht mit RUN starten, denn das bringt unweigerlich eine Fehlermeldung.

Adresse 23 und 24 ($17 und $18)

Zeiger auf die Adresse der letzten Zeichenkette im »Temporary String Stack«

Der Inhalt dieser 2 Byte zeigt auf den zuletzt benutzten Speicherplatz innerhalb der Adresse 22 bis 33. Das heißt, daß der Wert in 23 ($17) immer um 3 kleiner ist als der in 22 ($16), während der Wert in 24 ($18) eine Null ist.

Adresse 25 bis 33 ($19 bis $21)

Stapelspeicher für Angaben über vorläufige Zeichenketten

Das ist also der Speicherbereich, von dem in den beiden vorigen Abschnitten dauernd die Rede war. Ich gebe zu, »Descriptor Stack for Temporary Strings« drückt die Sache präziser aus als der deutsche Text.

Die Bedeutung eines »vorläufigen« Strings habe ich oben in der Beschreibung der Speicherzelle 22 erklärt.

Was ein Stapelspeicher (Stack) ist, entnehmen Sie bitte dem Texteinschub 6. Jeder der 3 Byte langen Angaben im Stack von 22 bis 33 enthält die Länge sowie die Anfangs- und Endadressen eines vorläufigen Strings, ausgedruckt als Verschiebung im Basic-Speicherbereich.

Adresse 34 bis 37 ($22 bis $25)

Verschiedene Zwischenspeicher

Diese vier Speicherzellen werden vom Basic-Übersetzer (Interpreter) für verschiedene Zwischenergebnisse und Flaggen benutzt, die aber dem Programmierer nichts nutzen.

Adresse 38 bis 42 ($26 bis $2A)

Arbeitsspeicher für arithmetische Operationen

Diese Speicherzellen werden von den Basic-Routinen bei der

Multiplikation und Division als »Notizblatt« verwendet. Auch die Routinen, welche die erforderliche Speichergröße beim Definieren eines Zahlenfeldes (Array) ausrechnen, benutzen diesen Bereich.

Adresse 43 und 44 ($2B und $2C)

Zeiger auf den Anfang der Basic-Programme im Speicher

Dieser Zeiger, in der Low-/High-Byte-Darstellung, gibt dem Basic-Übersetzer an, ab welcher Speicherzelle das Basic-Programm beginnt. Normalerweise ist diese Adresse fest vorgegeben. Beim C 64 zum Beispiel zeigt der Zeiger auf 2049 ($801). Beim VC 20 ist die Lage schon schwieriger, denn der Speicherbeginn hängt davon ab, welche Speichererweiterung eingesetzt ist. Die folgende Tabelle 3 gibt darüber Auskunft.

Tabelle 3: Beginn des Programmspeichers

C 64	**2049 ($801)**
VC 20 (GV)	**4097 ($1001)**
VC 20 (+3 K)	**1025 ($401)**
VC 20 (+ 8 K)	**4609 ($1201)**

Mit dem Befehl

```
PRINT PEEK (43) + PEEK
(44)*256
```

läßt sich der jeweilige Beginn des Programmspeichers leicht feststellen. Mit einem POKE-Befehl kann der Programmierer diese Anfangsadresse verändern. Wozu das gut ist, fragen Sie?

Anwendung #1:

Nun, wenn Sie zum Beispiel ein Maschinenprogramm mit einem Basic-Programm gemeinsam betreiben wollen, brauchen Sie einen Speicherbereich für das Maschinenprogramm, der vom Basic-Programm nicht belegt wird. Wir sprechen vom »Schützen des Maschinenprogramms vor dem Überschreiben durch das Basic«. Der Speicherbereich eines Maschinenprogramms ist immer bekannt. Nach seinem letzten Speicherplatz kann das Basic-Programm beginnen.

Die Verschiebung der Anfangsadresse erfolgt in vier Schritten:

1. Schritt: In den Speicherplatz vor dem neuen Basic-Bereich muß eine Null gePOKEt werden. Die Null dient zum Abgrenzen.
2. Schritt: Die Adresse der ersten Speicherzelle wird in die Low-/High-Byte-Darstellung umgerechnet. Ich verweise dazu auf die Erklärung dieses Vorgangs im Texteinschub Nr. 1.
3. Schritt: Das Low-Byte wird in die Speicherzelle 43, das High-Byte in die Zelle 44 gePOKEt.
4. Schritt: Die Operation muß unbedingt mit dem Befehl NEW abgeschlossen werden, um sicherzustellen, daß auch alle anderen Zeiger auf ihren Anfangszustand gesetzt werden.

Im folgenden kleinen Programm wird angenommen, daß der Speicher bis zur Adresse 5000 ($1388) durch ein Maschinenprogramm belegt ist. Das Basic-Programm kann daher ab 5002 ($138A) anfangen, denn in 5001 muß ja eine Null stehen. Die Adresse 5002 teilt sich auf in ein High-Byte von INT (5002/256) = 19 und ein Low-Byte von 5002-(19*256) = 138.

```
10 POKE 5001,0
20 POKE 43,138
30 POKE 44,19
40 NEW
```

Der Effekt einer solchen »Verbiegung« des Zeigers in 43 und 44 wird im Texteinschub Nr. 7 »Der sichtbare Basic-Speicher« demonstriert.

Neben der oben erwähnten Anwendung der Zeigerverbiegung gibt es noch andere Möglichkeiten:

Anwendung #2:

Christoph Sauer hat in seinem Kurs »Der gläserne VC 20« in Ausgabe 10/84 auf Seite 158 gezeigt, wie man mehrere Programme gleichzeitig im Speicher unterbringen und zwischen ihnen umschalten kann.

Anwendung #3:

Man kann zwei oder mehrere unabhängige Programme genau hintereinander in den Speicher bringen, um sie aneinander zu hängen, was dem im Commodore-Basic fehlenden Befehl MERGE entspricht. Dabei dürfen die Zeilennummern sich allerdings nicht überschneiden.

Anwendung #4:

Durch Hinaufschieben des Basic-Bereichs kann Platz geschaffen werden für selbstdefinierte Zeichen oder hochauflösende Grafik.

Die Speicherzellen-Paare von 45, 46 bis 55, 56 ($37 bis $38) zeigen auf weitere für Basic-Programme wichtige Speicherbereiche, die deswegen gemeinsam betrachtet werden sollten. Bild 5 stellt den Zusammenhang grafisch dar. In diesem Bereich werden alle Variablen eines Programms gespeichert. Zur Erinnerung:

Texteinschub Nr. 6
Was ist ein Stapelspeicher (Stack)?

Der normale Arbeitsspeicher des Computers, auf englisch »Random Access Memory« oder kurz RAM genannt, hat für jede Speicherzelle eine eigene Adresse, die beim Schreiben in den Speicher oder beim Lesen aus dem Speicher angegeben werden muß.

Als Analogie möge eine Aktenablage dienen, bei der jeder Akt (Brief, Papier, Zeichnung) in einen Ordner kommt, mit Nummer versehen.

Um einen Akt herauszuholen, muß man die Nummer (Adresse) kennen, unter der er abgelegt ist.

Ein Stapelspeicher, auf englisch »Stack« genannt, funktioniert wie eine Aktenablage, bei der jeder Akt einfach oben auf einen Stapel gelegt wird, daher der Name. Diese Ablage erfolgt ohne Kennzeichnung oder Nummer, einfach immer der Reihe nach.

Einen Akt kann man aus einem Stapelspeicher nicht beliebig herausholen, da immer nur der oberste Akt zugänglich ist.

Die Methode der Stapelspeicher bietet sich überall dort an, wo es auf die Reihenfolge der gespeicherten Daten ankommt. Basic merkt sich zum Beispiel der Reihe nach die Adressen, von denen aus mit GOSUB ein Unterprogramm angesprungen wird. Wenn mehrere GOSUBs hintereinander eingesetzt werden, liegt auf dem Stapel immer die letzte Absprungadresse bereit zum Rücksprung.

Ein Stapelspeicher hat demnach nur eine einzige Adresse, die sowohl zum Abspeichern als auch zum Auslesen dieselbe ist.

Voraussetzung eines Stapelspeichers ist natürlich eine Routine, welche alle gespeicherten Daten im Stapelspeicher um einen Platz weiterschiebt, wenn eine neue Information »oben auf den Stapel gelegt wird«.

Das Basic der Commodore-Computer verwendet mehrere dieser Stapelspeicher.

Die Programmiersprache Forth ist völlig auf dem Prinzip des Stapelspeichers aufgebaut.

Texteinschub Nr. 7
Der sichtbare Basic-Speicher

Wenn wir den Variablen A die Adresse des Speicherbeginns der Basic-Programme zuordnen und dann mit einer FOR..NEXT-Schleife den Inhalt dieser und der nächsten 100 Speicherplätze ausdrucken, sehen wir in dezimaler Darstellung die ersten 101 Zahlenwerte, mit denen der Computer ein Basic-Programm speichert.

Ein Verbiegen des Zeigers in Speicherzelle 43/44 kann auf diese Weise in seiner Wirkung sichtbar gemacht werden.

Als Demo-Programm wähle ich zwei Zeilen, welche die Zahlen 1 bis 9 und die Buchstaben A bis I ausdrucken.

```
 10 PRINT "123456789"
 20 PRINT "ABCDEFGHI"
100 A=2049 : REM*C 64
      4097 : REM*VC 20 ohne Erweiterung
      1025 : REM*VC 20 mit 3 KByte
      4609 : REM*VC 20 mit 8 KByte oder mehr
110 PRINT CHR$(147)
```

Zeile 100 definiert den Speicheranfang. Zeile 110 löscht den Bildschirm.

```
120 FOR J=A TO A+100
130 PRINT PEEK (J);
140 NEXT J
```

Die Befehle in den Zeilen 120 bis 140 drucken den Inhalt der ersten 101 Zellen dieses Basic-Programms aus. SAVEn Sie bitte dieses kleine Programm, denn wir brauchen es noch einmal. Dann geht es los mit RUN. In Bild 3 ist der Bildschirm-Ausdruck des VC 20 mit 8 KByte dargestellt, der des C 64 zeigt praktisch dieselbe Information.

Überspringen Sie bitte zunächst die ersten beiden Zahlen. Die dritte und vierte Zahl ist 10 und 0. Das ist (als Low- und High-Byte) die Nummer der ersten Zeile des Basic-Programms. Dann folgt 153, das ist der interne Codewert für PRINT. Diese Codes für alle Basic-Befehlswörter heißen »TOKEN«, sie sind im Texteinschub Nr. 32 angegeben.

Die nächste Zahl auf dem Bildschirm ist die 34, sie ist der ASCII-Code für den Gänsefuß. Danach folgen in aufsteigender Reihenfolge die ASCII-Codes der Ziffern 1 (48) bis 9 (57). Danach sehen Sie wieder den Gänsefuß (34). Schließlich kommt eine Null als Abstandszeichen zur nächsten Basic-Zeile.

Machen Sie bitte folgendes Experiment: Ausgehend von der Adresse der ersten auf dem Bildschirm ausgedruckten Speicherzellen - zum Beispiel 4609 beim VC 20 mit 8 KByte - zählen Sie die Zellen weiter bis zur Abgrenzungs-Null. In meinem Beispiel steht die Null in Zeile 4625. Das heißt, daß die nächste Basic-Zeile in 4626 anfängt. Und das ist genau die Zahl, die in den ersten beiden Zellen steht, die wir vorhin übersprungen haben; in meinem Beispiel steht da 18 18. Machen wir die Probe: 18 + 256 * 18 = 4626.

Jede Basic-Zeile im Speicher beginnt also mit der Adresse der nächsten Zeile (sie heißt Koppeladresse) und endet mit einer Null.

Ab 4626 folgt dann die nächste Koppeladresse, danach mit 20 0 die Zeilennummer, und Sie erkennen jetzt sicher die Codes der Angaben von Zeile 20 wieder.

So, jetzt wollen wir den Zeiger in 43 und 44 verbiegen. Ich schlage vor, daß wir den Basic-Beginn um zehn Adressen höher schieben. Sie müssen jetzt die in Zeile 100 oben verwendete Zahl für A in die High-/Low-Byte-Darstellung umrechnen und das Low-Byte um 10 erhöhen. Dieses Zahlenpaar POKEn wir in die Zellen 43 und 44. Vorher müssen wir aber noch in Zelle (A + 10) -1 eine Abstands-Null POKEn.

Wir geben diese Befehlssequenz im Direktmodus ein:

☐ für den C 64:

```
POKE 2058,0:POKE 43,11:POKE 44,8:NEW
```

☐ für den VC 20 (GV):

```
POKE 5006,0:POKE 43,143:POKE 44,19:NEW
```

☐ für den VC 20 (= 3 KByte):

```
POKE 1034,0:POKE 43,11:POKE 44,4:NEW
```

☐ für den VC 20 (> 8 KByte)

```
POKE 4618,0:POKE 43,11:POKE 44,18: NEW
```

Jetzt ist der Anfang des Basic-Speichers versetzt. Um das zu prüfen, geben wir das kleine Programm von oben nochmal ein und lassen es mit RUN laufen. Der resultierende Bildschirmausdruck ist in Bild 4 dargestellt.

Die ersten Zahlen sind genauso wie vorher. Es sind auch die Reste von vorher, da wir den Speicher nicht auf Null gesetzt haben. Aber zählen Sie bitte die ersten zehn Adressen hoch. Da finden Sie unser Programm von vorhin genau wieder, beginnend mit der Abstands-Null. Aber Vorsicht, lassen Sie sich nicht verwirren, denn die Koppeladressen sind natürlich jetzt auch jeweils um 10 höher. Aber hinter den Koppeladressen finden wir wieder unser Programm, in gleicher Weise dargestellt wie beim ersten Mal. Da der Zeiger in 43 und 44 von allen entsprechenden Routinen des Übersetzers und des Betriebssystems abgefragt wird, läuft ein verschobenes Programm fehlerfrei, solange natürlich der Zeiger nicht wieder verändert wird.

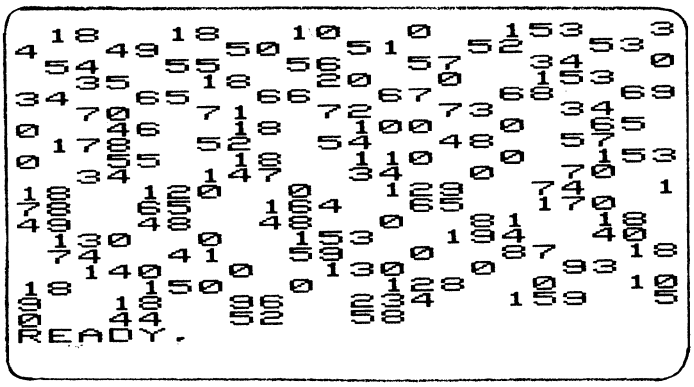


Bild 3. Ausdruck vom VC 20 mit 8 KByte Speichererweiterung

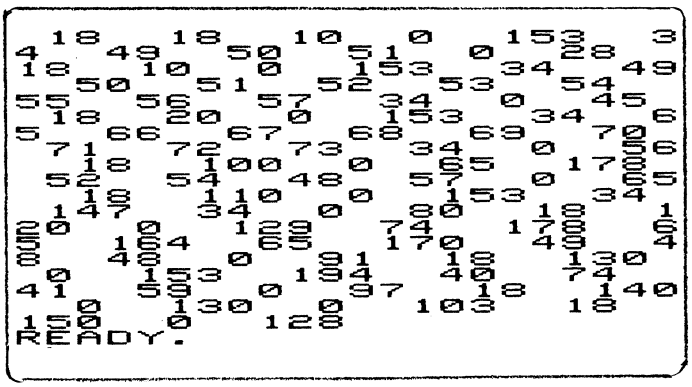


Bild 4. Neuer Bildschirmausdruck

Wir unterscheiden zwischen »normalen« Variablen (numerische und String-Variable) und Feld-Variablen (Arrays). Dabei ist wichtig zu wissen, daß ein Basic-Programm während des Eintippens oder Einladens von Disk beziehungsweise Kassette in den 1. Block kommt. Während des Programmlaufs werden alle normalen Variablen in den 2. Block geschrieben, alle Felder (Arrays) in den 3. Block und schließlich der Text der Zeichenketten (Strings) sozusagen rückwärts vom Ende des Arbeitsspeichers in den 4. Block. Je nach Größe des Programms und nach Anzahl der Variablen wandern die Blockgrenzen nach oben beziehungsweise die von Block 4 nach unten. Wenn sie sich treffen beziehungsweise überschneiden, gibt es »OUT OF MEMORY«.

Diese Blockbewegung ist in Bild 5 durch die Pfeile dargestellt.

Adresse 45 und 46 ($2D und $2E)

Zeiger auf die Anfangsadresse des Speicherbereichs für Variable

Dieser Zeiger, in der Low/High-Byte-Darstellung, gibt dem Basic-Interpreter an, ab welcher Speicherzelle die Variablen eines Basic-Programms gespeichert sind. Da die Variablen direkt an das Basic-Programm anschließen, zeigt dieser Zeiger natürlich gleichzeitig auf das Ende des Basic-Programms.

Es muß betont werden, daß es sich nur um den Bereich der »normalen« Variablen handelt, also nicht um Felder (Arrays). Anders als der Zeiger in 43 und 44, der auf fest definierte Speicherzellen zeigt, liegt der Zeiger für den Variablen-Beginn nicht fest. Je nach Länge des Programms wandert er nach oben.

Sobald ein Programm eingetippt oder aus einem externen Speicher (Diskette, Kassette) eingelesen ist, wird der Zeiger in 45 und 46 durch RUN auf ein Byte hinter das Programmende gesetzt und alle Variablen werden in der Reihenfolge ihres Auftretens gespeichert. Da normalerweise die Länge eines Basic-Programms während des Ablaufs konstant bleibt, werden die Variablen in ihrer Position auch nicht gestört.

Das bedeutet, daß sie sowohl vom Programm als auch vom Programmierer nach einer Unterbrechung abgefragt werden können. Nur wenn das Programm modifiziert wird, wandert der Zeiger zusammen mit den Variablen entsprechend weiter.

Denselben Effekt wie das oben erwähnte RUN haben übrigens auch die Befehle NEW, CLR und LOAD. Eine Ausnahme bildet das LOAD innerhalb eines Programms, welches den Zeiger nicht zurücksetzt. Dadurch wird ein Aneinanderhängen von mehreren Programmen samt Variablen-Weiterverwendung unter bestimmten Voraussetzungen ermöglicht.

Die Bearbeitung der Variablen durch das Basic-Programm und die daraus resultierenden Kochrezepte für den Programmierer sind im Texteinschub Nr. 8 »Normale Variable in BASIC« separat erläutert.

Die verschiedenen Typen der Variablen und ihre Darstellung

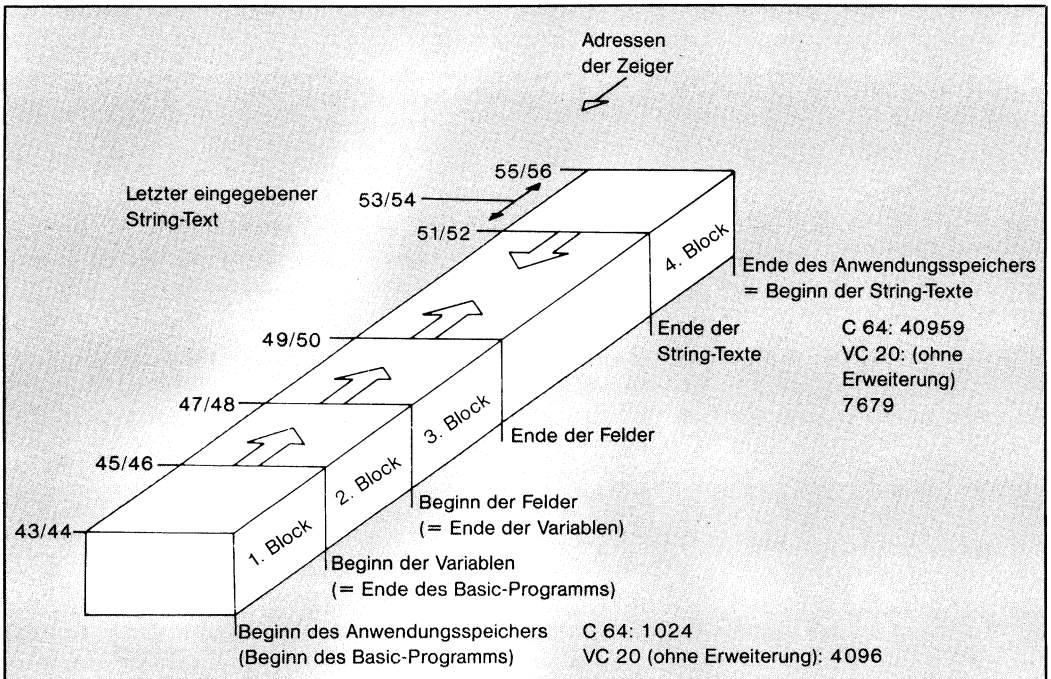


Bild 5. Eine schematische Aufteilung des Speichers des C64/VC20

im Speicher finden Sie im 64'er, Ausgabe 10/84, Seite 157 und noch ausführlicher in Ausgabe 11/84, Seite 124, dargestellt und erklärt.

Für diejenigen Leser, welche kein Monitor- beziehungsweise Disassembler-Programm haben oder benutzen können, ist im Texteinschub Nr. 9 »Darstellung der normalen Variablen im Speicher« eine kleine Anleitung gegeben, wie sie die Variablendarstellung mittels Basic anschauen können.

Adresse 47 und 48 ($2F und $30)

Zeiger auf die Anfangsadresse des Speicherbereichs für Felder (Arrays)

Dieser Zeiger, in der Low-/High-Byte-Darstellung, gibt dem Basic-Übersetzer (Interpreter) an, ab welcher Speicherzelle die Felder (Arrays) eines Basic-Programms gespeichert

Texteinschub Nr. 8
Normale Variable in Basic

Alle Daten, die in einem Basic-Programm nicht in Form von READ-DATA-Anweisungen vorkommen, werden als »Variable« unmittelbar nach dem Basic-Programm abgespeichert. Wir unterscheiden dabei zwei Typen:
- normale Variable
- Felder (Arrays)

Wir betrachten hier nur die »normalen« Variablen.

Sie erscheinen in dem Speicherbereich, dessen Beginn durch den Zeiger in den Zellen 45 und 46 und dessen Ende durch den Zeiger in 47 und 48 angegeben wird, in derselben Reihenfolge, in welcher sie während des Ablaufes des Basic-Programms auftreten. Wenn Basic dann auf eine der bereits definierten (und abgespeicherten) Variablen zurückgreifen soll, muß es den gesamten Variablenbereich von Anfang an absuchen, bis es den Namen der gesuchten Variablen gefunden hat. Wenn diese Variable ganz am Ende des Bereiches steht, kann dieser Suchprozeß recht lange dauern.

Regel 1:

Häufig vorkommende Variable sollen am Anfang des Variablenbereichs stehen. Das wird dadurch erreicht, daß sie als erste Variable in einem Programm »definiert« werden. Falls sie erst später im Programm gebraucht werden (aber dann häufig), werden sie trotzdem am Anfang des Programms angegeben, notfalls mit einem beliebigen Wert, der später dann keine Rolle mehr spielt und ersetzt wird. Man nennt das einen »Dummy«-Wert.

Die Felder-Variablen stehen direkt nach den »normalen« Variablen. Auch hier kann der gewiefte Programmierer Gutes tun. Wenn nämlich nach einer Definition eines Feldes später im Programm noch normale Variable dazukommen, ist natürlich zuerst kein Platz für sie da. Das Betriebssystem des Computers muß erst alle Felder-Variablen weiterschieben, bevor die Neuankömmlinge in dem dadurch erweiterten Variablenbereich gespeichert werden können. Auch das kostet unnötig viel Zeit.

Regel 2:

Alle normalen Variablen sollen als erste in einem Programm definiert werden. Wer also drauflos programmiert, sollte zumindest am Ende das Programm so umbauen, daß diese simple Regel erfüllt wird.

Texteinschub Nr. 9
Darstellung der normalen Variablen im Speicher

Die normalen Variablen kommen in drei Arten vor:
- ganzzahlige Variablen
- Gleitkomma-Variablen
- String-Variablen (Zeichenketten)

Der Unterschied zwischen den drei Typen ist in den Commodore-Handbüchern gut erklärt, und ich verzichte hier auf eine Wiederholung. Ich will vielmehr direkt zeigen, wie die Variablen im Speicher abgelegt sind.

Wir können den Speicher direkt sichtbar machen.

Einmal geht das in Maschinencode mittels eines Monitors beziehungsweise Disassemblers.

Zum anderen aber geht das auch in Basic und zwar mit folgendem Trick, den ich Th. und M.L. Beyer (MC 10/1983) abgeschaut habe.

Wir verlegen den Beginn des Basic-Variablenspeichers einfach auf den Beginn des Bildschirmspeichers. Auf diese Weise können wir zwar kein vernünftiges Programm laufen lassen, aber alle direkt eingegebenen Variablen-Definitionen werden sofort sichtbar, weil sie eben im Bildschirmspeicher stehen.

Wir erreichen die Verlegung des Speichers durch »Verbiegen« der Zeiger in den Zellen 45 und 46 und 47 und 48. Die Bedeutung dieser Zeiger ist ja im Kurs erklärt.

Die Speicherverlegung beziehungsweise die Methode dazu ist für den C 64 anders als für den VC 20.

■ VC 20:

Alle Angaben gelten für den VC 20 ohne Speichererweiterung, also ziehen Sie bitte alle Speichermodule heraus. Der Speicherbereich für Programme und deren Variablen beginnt jetzt ab Adresse 4096, das ist Block 1 im Bild 5. Der Bildschirmspeicher beginnt ab 7680. Wir verlegen jetzt den Bildschirmspeicher in den Block 1, so daß er ebenfalls ab Adresse 4096 beginnt. Danach müssen wir noch eine Farbe - am besten Schwarz - in den Farbspeicher POKEn, der in dieser neuen Konfiguration von 37888 bis 38399 liegt. Warum das so ist, erklärt Christoph Sauer in seinem Aufsatz »Der gläserne VC 20«, Teil 4, im 64'er 1/85, Seite 131.

Das High-Byte der Adresse, in welcher der Bildschirmspeicher

beginnt, steht in der Speicherzelle 648. Sie können das jederzeit mit PRINT PEEK(648) nachprüfen. Umgekehrt können wir eine Zahl hineinPOKEn, wodurch der Bildschirmspeicher verschoben wird. In unserem Fall erhalten wir das High-Byte für 4096 durch 4096/256 = 16.

Machen Sie jetzt bitte folgende Schritte:

1) direkt eingeben: `POKE 648,16(RETURN),`

2) RUN/STOP und RESTORE drücken, bis der Cursor wieder da ist,

3) direkt eingeben:

```
FOR J = 37888: TO 38399: POKE J,0: NEXT J (RETURN),
```

4) mit der DELETE-Taste (nicht mit CLR !) den ganzen Text des Bildschirms löschen,

5) mit dem Cursor etwa acht Zeilen nach unten gehen,

6) mit der Commodore- und SHIFT-Taste zusammen auf die Groß- und Kleinschrift umstellen.

Schritt 1 und 3 habe ich oben schon erklärt. Schritt 4 ist nicht absolut notwendig, aber ein leerer Bildschirm ist für uns besser. Die CLR-Taste würde Schritt 3 zunichte machen. Schritt 5 erlaubt uns, weiter unter auf dem Bildschirm Variablen einzugeben, ohne den oberen Teil vollzuschreiben. Schritt 6 schließlich erleichtert das Erkennen der Variablen-Darstelllung.

■ C 64:

Beim C 64 beginnt der Bildschirmspeicher ab 1024. In Low-/High-Byte-Darstellung ist das 0/4 (1024/256=4, Rest 0). Geben Sie bitte direkt ein:

```
POKE 46,4 :POKE 48,4
```

Das Low-Byte in 45 und 47 können wir weglassen, da es ja in beiden Fällen 0 ist. Diese Methode gilt für alle neueren C 64, bei denen direkt ein Zeichen in den Bildschirmspeicher gePOKEt werden kann, ohne sich um die Zeichenfarbe kümmern zu müssen. Es gibt noch einige C 64 mit älteren Betriebssystemen, bei denen die Zeichenfarbe auch angegeben werden muß. Hier gilt:

```
FOR J = 0TO 1000:POKE 55296+3,14:NEXT J
```

■ Alles weitere gilt für beide Computertypen gleich

Wenn Sie jetzt den Bildschirm löschen, den Cursor ungefähr in die Mitte des Bildschirms fahren und wiederum direkt eingeben:

`VARIABLE = 3` und die RETURN-Taste drücken,

dann erscheinen oben sieben Zeichen. Bitte schalten Sie mit der SHIFT- und Commodore-Taste auf den zweiten Zeichensatz um, jetzt können wir besser lesen.

Aus anderen Kursen wissen Sie wahrscheinlich, daß Variable mit 7 Byte dargestellt werden. In der Tat sehen wir oben die ersten beiden Buchstaben des Variablennamens VA und fünf weitere Zeichen. Wir wollen aber systematisch vorgehen und uns zuerst die ganzzahligen Variablen anschauen.

Ganzzahl-Variable

Wiederholen Sie bitte den Vorgang (Löschen, Cursor auf Mitte, 2. Zeichensatz). Jetzt geben Sie eine Ganzzahl-Variable ein:

```
VA%=3
```

Nach RETURN sehen wir als erstes Zeichen ein reverses V, dann ein reverses A, den Klammeraffen @, das kleine c und nochmals drei @. Die beiden ersten Zeichen des Variablennamens (besteht er nur aus einem Zeichen, wird mit einer 0 aufgefüllt) werden mit ihrem ASCII-Code eingegeben, zu dem bei Ganzzahl-Variablen zur Kennzeichnung einer solchen die Zahl 128 addiert wird.

Schauen Sie in einer ASCII-Tabelle (64'er, Ausgabe 7/84) nach: Das V hat 86, um 128 erhöht gibt das 214. Wir arbeiten hier aber im Bildschirmspeicher, der die Zahlen auf seine eigene Weise interpretiert, nämlich als Bildschirmcode. Der Bildschirmcode-Tabelle entnehmen wir das Zeichen für den Wert 214 und das ist das invertierte V. Für das A können Sie das selbst nachvollziehen.

Also: In unserer Darstellung erkennen wir Ganzzahl-Variable an den invertierten Zeichen des Namens.

Das 3. und 4. Zeichen sind das High- und Low-Byte des Variablenwertes und zwar im Bildschirmcode. In unserem Beispiel der 3 ist das High-Byte 0, also der Klammeraffe @, das Low-Byte 3, also das c. Die restlichen drei Byte sind mit 0 aufgefüllt.

Wenn Sie mit dem Cursor auf die 3 fahren, es mit einer 5 überschreiben und RETURN drücken, verwandelt sich das c in ein e. Beim Überschreiben mit 255 erscheint als 4. Byte das Zeichen für den Bildschirmcode 255. Beim Überschreiben mit 257 ändern sich beide Bytes. Das 3. (High-)Byte springt auf a (=1), das 4. (Low-)Byte ebenfalls auf a. Nun, 1 * 256+1 = 257.

Während, wie bewiesen, das Low-Byte von 0 bis 255 gehen kann, sind beim High-Byte nur Werte zwischen 0 und 127 zugelassen. Die Werte ab 128 signalisieren negative Zahlen. Probieren Sie es aus:

```
127 * 256+255=32767
```

Ein Überschreiben mit 32767 resultiert in einer Darstellung der Zeichen für den Bildschirmcode 127 und 255. Der Wert 32768 wird nicht mehr akzeptiert. Dasselbe machen wir noch schnell für negative Zahlen.

Überschreiben Sie bitte die letzte Zahl mit 0. Wie zu erwarten war, sind Byte 4 und 5 jetzt 0 (Klammeraffe).

Wenn Sie jetzt mit -1 überschreiben, erscheint für beide Bytes das Zeichen mit dem Bildschirmcode 255. Bei -2 sehen wir die Zeichen mit den Code-Werten 255 und 254.

Sie sehen also, daß die negativen Zahlen für ganzzahlige Variable sozusagen vom Ende der Tabelle her dargestellt werden, wobei die höchste negative Zahl wieder 32767 ist. Diese »Rückwärtszählung« ist bedingt durch die Methode der negativen Zahlendarstellung im Zweierkomplement. Der Platz und die Gelegenheit verbieten es mir, näher darauf einzugehen. Aber ich glaube, unser kleines Experiment hat Ihnen zumindest von der Darstellung her den Zusammenhang gezeigt. In Bild 6 ist diese Darstellung der ganzzahligen Variablen im Speicher wiedergegeben.

1	2	3	4	5 6 7
Erstes	Zweites	High-	Low-	
Zeichen des Variablen-Namens (ASCII-Wert + 128)		Byte des Variablenwertes		0 0 0

Bild 6. So stehen ganzzahlige Variable im Speicher

Gleitkomma-Variable

Ich hoffe, Sie verzeihen mir, wenn ich diese Darstellung an dieser Stelle überspringe. Sie ist nämlich nicht ganz leicht zu verstehen, und ich möchte sie lieber dann im Detail erklären, wenn wir zur Diskussion der Speicherzellen 97 bis 101, nämlich des Gleitkomma-Akkumulators kommen. Da geht es in einem Stück. Als Vorgeschmack gebe ich jetzt in Bild 7 nur die Zusammenfassung an.

1	2	3	4 5 6 7
Erstes	Zweites		
Zeichen des Variablen-Namens (ASCII-Wert)		Exponent + 129	Mantisse mit Genauigkeit von 32 Dualstellen, 1. Bit des 1. Bytes ist das Vorzeichen

Bild 7. Der FAC = Gleit- oder Fließkomma-Akkumulator

String-Variable

Zuerst ist es erforderlich, den Computer in den Anfangszustand zurückzusetzen. Wenn Sie einen RESET-Schalter haben, bitte diesen drücken, sonst aber aus- und einschalten. Wir geben nach Löschen des Bildschirms in der unteren Hälfte direkt ein:

```
ZX$="A" <RETURN>
```

Wir erhalten ein Z, ein invertiertes X, ein kleines a, ein grafisches Zeichen, eine Leerstelle und zwei Klammeraffen.

Fahren Sie bitte jetzt mit dem Cursor auf das A und ändern den String um in BC. Nach RETURN verwandelt sich das a in das b,

das 4. Zeichen ändert sich ebenfalls. Die ersten beiden Zeichen sind also wieder der Name der Variable.

Um zu kennzeichnen, daß es eine String-Variable ist, erscheint das 2. Zeichen des Namens invertiert. Wie oben entsteht es dadurch, daß zum ASCII-Code die Zahl 128 addiert wird. Diese Zahl wird aber wieder als Bildschirmcode interpretiert und entsprechend angezeigt (vergleichen Sie es mit den ASCII- und Bildschirmcode-Tabellen).

Das 3. Zeichen gibt die Länge des Strings an, also im ersten Fall mit a (=1 im Bildschirmcode), im 2. Fall mit b (=2). Zeichen 4 und 5 geben als Low- und High-Byte die Adresse an, bei der begonnen wird, den Text des Strings zu speichern. Das können wir nachprüfen.

Wir hatten die beiden Fälle:

1. ZX $ = "A"

4. Zeichen: (Bildschirmcode: 255) und 5. Zeichen: (Bildschirmcode 156) ergibt als Adresse 40959.

2. ZX$ = "BC"

4. Zeichen: (Bildschirmcode 253) und 5. Zeichen: (Bildschirmcode 156) ergibt als Adresse 40957.

Der Text der Zeichenketten wird am Ende des Arbeitsspeichers (40959 beim C 64, 7679 beim VC 20 ohne Erweiterung) abgelegt und zwar von hinten nach vorn.

Mit `PRINT PEEK(40957);PEEK(40958);PEEK(40959)` drucken wir den Inhalt dieser Speicherzellen aus und erhalten: 66 67 65. Im ASCII-Code ist das: B C A. Die Zusammenfassung für String-Variable (Bild 8) sieht so aus:

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6 7</td></tr>
<tr><td>Erstes</td><td>Zweites</td><td rowspan="3">Anzahl der Zeichen des Strings</td><td>Low-</td><td>High-</td><td rowspan="3">0 0</td></tr>
<tr><td colspan="2">Zeichen des Variablen-Namens</td><td colspan="2" rowspan="2">Byte der Adresse, ab welcher der Text des Strings abgespeichert ist</td></tr>
<tr><td>ASCII-Wert</td><td>ASCII-Wert+128</td></tr>
</table>

Bild 8. String-Variable

sind. Was Felder sind und wozu sie gebraucht werden, ist im Texteinschub Nr. 10 kurz erläutert. Da die Felder direkt nach den normalen Variablen gespeichert werden, zeigt dieser Zeiger natürlich gleichzeitig auf das Ende des Speichers für normale Variablen.

Durch POKEn einer Adresse in die Speicherzellen 47 und 48 kann der Speicherbereich am Anfang eines Programms beinahe beliebig verschoben werden. Beinahe deswegen, weil die Verschiebung im Zusammenhang mit den anderen Bereichen (siehe Bild 5) einen Sinn haben muß. Im übrigen gilt für diesen Zeiger dasselbe, was schon für den Zeiger in 45 und 46 gesagt worden ist. Die Darstellung der Feld-Variablen selbst kann mit der Methode angesehen werden, die im Texteinschub Nr. 11 erklärt ist.

Wie aus den Erklärungen hervorgeht, wird bei Feldern mit Zeichenketten (Strings) in dem von Zeiger 47 und 48 bezeichneten Speicherbereich nur die Definition beziehungsweise die Dimensionierung gespeichert. Die eigentlichen Zeichenketten stehen wie bei den normalen Variablen im vierten Block, vom Speicherende rückwärts angeordnet.

Adresse 49 und 50 ($31 und $32)

Zeiger auf die Endadresse (+1) des Speicherbereichs für Felder (Arrays)

Der Inhalt dieser Speicherzellen zeigt auf die Adresse, wo der Speicherbereich für Felder aufhört. Wie aus Bild 5 hervorgeht, werden die Zeichenketten vom Ende des verfügbaren RAM-Speichers rückwärts gespeichert. Man kann also auch sagen, daß der Zeiger in 49 und 50 die letzte mögliche Adresse für Zeichenketten angibt. Wenn in einem Programm neue Variablen definiert werden, rutscht diese Adresse weiter nach oben und nähert sich dem Ende der Zeichenketten, die durch den Zeiger in 51 und 52 angegeben wird.

Wenn sich die Speicherbereiche der Felder und Zeichenketten berühren, bleibt der Computer stehen und führt die »Garbage Collection« (Müllabfuhr) durch – ein Prozeß, in dem nicht mehr gebrauchte Zeichenketten entfernt und der Zeichenketten-Speicher reduziert wird. Ist danach immer noch kein Platz, wird OUT OF MEMORY gegeben.

Der Befehl FRE löst immer eine solche Garbage Collection aus und gibt dann die Differenz zwischen den Adressen in den Zeigern 49 und 50 und 51 und 52 als verbleibenden, noch verfügbaren, Speicherbereich aus.

Adresse 51 und 52 ($33 und $34)

Zeiger auf die untere Grenze des Speicherbereichs für den Text der Zeichenketten-Variablen

Der Inhalt dieser Speicherzellen zeigt in Low-/High-Byte-Darstellung auf das jeweilige untere Ende (siehe Bild 5) des Textspeichers von Zeichenketten. Er bezeichnet aber zugleich auch das obere Ende des frei verfügbaren RAM-Bereichs. Das entsteht dadurch, daß der Text der Zeichenketten vom Ende des RAM-Bereichs nach unten gespeichert wird. In Bild 5 ist das durch den Pfeil dargestellt.

Beim Einschalten des Computers und nach einem RESET wird dieser Zeiger auf das oberste Ende des RAM-Bereichs gesetzt. Beim C 64 ist das 40960 ($A000). Beim VC 20 hängt es von den eingesetzten Speichererweiterungen ab, ohne Erweiterung ist die Adresse 7680 ($1E00).

Der Befehl CLR setzt den Zeiger auf die Adresse, welche durch den Zeiger in den Speicherzellen 55 und 56 als das Ende des Basic-Speichers angegeben wird. Wozu das dient, erkläre ich Ihnen bei der Beschreibung dieses Zeigers weiter unten.

Adresse 53 und 54 ($35 und $36)

Zeiger auf die Adresse der zuletzt eingegebenen Zeichenkette

In diesen Speicherplätzen steht die Adresse (im vierten Block, siehe Bild 5) der Zeichenkette, die als letzte von Routinen (Programme, Direkteingabe) zur String-Manipulation abgespeichert worden ist. Mit dem folgenden kleinen Programm können Sie das genau sehen:

```
10 PRINT PEEK(53)+
256*PEEK(54),
20 PRINT PEEK(51)+256*PEEK
(52)
30 INPUT A$
40 GOTO 10
```

Zeile 10 druckt uns zuerst (links) den Zeiger auf die zuletzt eingegebene Zeichenkette aus, Zeile 20 rechts daneben den Zeiger auf die untere Speichergrenze der Zeichenketten. Zeile 30 fordert zur Eingabe einer Zeichenkette auf.

Wenn Sie bei frisch eingeschaltetem Computer das Programm starten, sehen Sie eine 0 (=vorher noch kein String eingeben) und daneben die Adresse dezimal 40960 (C 64) beziehungsweise dezimal 7680 (VC 20 ohne Erweiterung). Wenn Sie auf das Fragezeichen des INPUT hin zum Beispiel ein A eintippen, erhalten Sie links den vorigen Wert von rechts und rechts jetzt eine um 1 kleinere Zahl. Eine weitere Eingabe von zum Beispiel XXXXX schiebt die alte rechte Zahl nach links und die neue wird um die Anzahl der Zeichen, also 5, verringert.

Adresse 55 und 56 ($37 und $38)

Zeiger auf das Ende des für Basic-Programme verfügbaren Speichers

Dieser Zeiger, in der Low-/High-Byte-Darstellung, gibt dem Basic-Übersetzer an, welches die höchste von Basic verwendbare Speicheradresse ist. Wie aus Bild 5 ersichtlich, ist diese Adresse zugleich der Anfang der als Variable abgespeicherten Zeichenkette (Strings).

Normalerweise ist diese Adresse fest vorgegeben. Die folgende Tabelle 4 gibt darüber Auskunft:

Tabelle 4.
Ende des Programmspeichers

	Adresse	Zeiger in 55	56
C 64	40960	0160	
VC 20 (Grundv.)	7680	030	
VC 20 (+3 KByte)	7680	030	
VC 20 (+8 KByte)	16384	064	

VC 20		
(+16 KByte)	24576	096
VC 20		
(+24 KByte)	32768	0128

Beim Einschalten des Computers überprüft das Betriebssystem den gesamten RAM-Speicher, bis es zur ersten ROM-Speicherzelle kommt, setzt den Zeiger in 55 und 56 auf diese Adresse und druckt den bekannten Kopf mit der verfügbaren Speicherangabe auf den Bildschirm.

Normalerweise wird dieser Zeiger nicht geändert.

Es gibt aber zwei Gelegenheiten, bei denen eine Änderung dieses Zeigers sinnvoll beziehungsweise notwendig ist.

Anwendung 1:

Es kommt oft vor, daß der gesamte Speicher nicht ausschließlich für Basic benötigt wird, sondern daß ein freier Speicherbereich geschaffen wird, um zum Beispiel Maschinenprogramme, selbst definierte Zeichen oder hochaufgelöste Grafik unterzubringen, die aber nicht vom Basic-Programm überschrieben werden können.

Bei der Besprechung der Zeiger in 43 und 44 haben wir das auch schon gemacht, allerdings durch »Hochschieben« des Speicheranfangs. Mit dem Zeiger in 55 und 56 erreichen wir denselben Effekt, diesmal durch »Herunterdrücken« des Speicherendes. Gegenüber den vier Schritten beim Hochschieben ist das Herunterdrükken einfacher. Mit dem Befehl:

```
POKE 56,PEEK(56)-1:CLR
```

schieben wir das Speicherende um 256 Byte nach unten, egal für welchen Computer und welche Speichererweiterung. Mit -2 verschiebt sich das Ende um 512, mit -4 um 1 024 Byte (also 1 KByte) nach unten. Wenn Sie eine feinere Verschiebung als Vielfache von 256 benötigen, kommen Sie mit dem High-Byte in 56 allein nicht aus, sondern Sie müssen auch einen entsprechenden Wert in 55 hineinPOKEn.

Der Befehl CLR ist notwendig, denn er setzt den Zeiger der Zellen 51 und 52 (siehe dort), das heißt das untere Ende des Speicherbereichs für Zeichenketten auf dieselbe Adresse wie Zeiger 55 und 56. Dadurch wird erzwungen, daß die Zeichenkette sozusagen als Ausgangslage unterhalb des heruntergedrückten Speicherendes abgelegt wird.

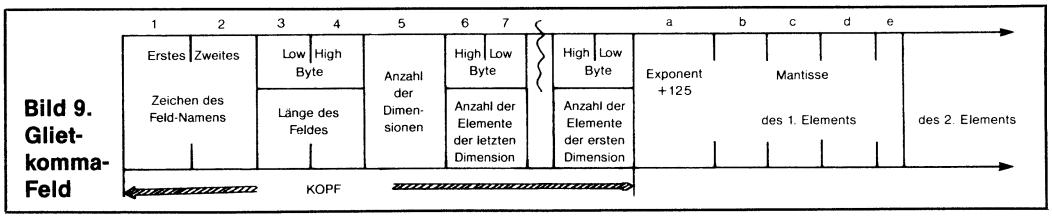


Bild 9. Gliet-komma-Feld

Anwendung 2:

Über den User-Port (Steckerleiste an der Rückseite, neben dem Datasetten-Anschluß) können VC 20 und C 64 mit anderen Geräten verbunden werden. Der Datentransfer über diese Verbindung - sie heißt RS232-Schnittstelle - muß allerdings programmiert werden. Diese RS232-Schnittstelle hat die Gerätenummer 2 (so wie der Drucker Nummer 4 und das Diskettengerät die Nummer 8 hat).

Wenn nun ein Gerät Nummer 2 mit einem OPEN-Befehl angewählt wird, wird automatisch der Zeiger in 55 und 56 und der Zeiger in 643 um 512 Byte heruntergedrückt, um je einen Eingangs- und Ausgangspufferspeicher zu erzeugen. Da der Inhalt dieser Pufferspeicher alle Variable in diesen 512 Byte überschreiben würde, wird auch der CLR-Befehl automatisch gegeben.

Es gilt daher als Vorschrift, daß bei RS232-Verbindungen zuerst der Datenkanal durch OPEN eröffnet werden muß, bevor Variable, Felder und Zeichenketten definiert werden.

Adresse 57 und 58 ($39 und $3A)

Nummer der laufenden Basic-Programmzeile

Diese Speicherzellen enthalten die Zeilennummer in Low-/High-Byte-Darstellung derjenigen Basic-Anweisung, welche gerade ausgeführt wird.

Ein kurzes Programm macht das deutlich:

```
10 PRINT "ZEILE 10",
PEEK(57)+256*PEEK(58)
20 A=3:PRINT A,PEEK(57)+
256*PEEK(58)
30 B=5:PRINT B,PEEK(57)+
256*PEEK(58)
40 PRINT A*B,PEEK(57)+
256*PEEK(58)
```

In jeder Zeile wird zuerst etwas gePRINTet, nämlich Text, Variable und ein Rechenresultat. Durch das Komma getrennt wird in der 2. Bildschirmhälfte (VC 20) beziehungsweise Bildschirmviertel (C 64) der Inhalt der Speicherzellen 57 und 58 ausgedruckt. Das Resultat zeigt

Texteinschub Nr. 10
Felder in Basic

Zur Wiederholung: Es gibt zwei Arten von Variablen, normale Variable und Felder. Jede der beiden Arten ihrerseits kann aus Gleitkomma-Zahlen, ganzen Zahlen oder Zeichenketten bestehen.

Eine normale Variable kann immer nur einen Wert haben, ein Feld enthält gleichzeitig viele Werte, alle unter demselben Variablen-Namen.

Wir können uns ein Feld mit dem Namen KARLSTRASSE als eine Liste vorstellen, in der jedes Element zwar auch den Namen Karlstraße hat, sich aber von den anderen Elementen durch eine eigene Hausnummer unterscheidet. Jede Variable in einer Hausnummer hat einen bestimmten Wert.

Während eine normale Variable einfach mit A=3 einen Wert zugewiesen bekommt, muß ein Feld erst definiert werden, nämlich wie viele Elemente es enthält. Wir machen das mit dem Befehl

```
DIM KARLSTRASSE (12)
```

Dieses Feld hat 13 Elemente (von 0 bis 12). Jedem Element kann nun ein Variablenwert zugewiesen werden durch

```
KARLSTRASSE (0)=25
KARLSTRASSE (1)=56
```

Das Feld KARLSTRASSE hat in der Klammer nur eine Zahl, man sagt, es hat nur eine Dimension.

Ein zweidimensionales Feld entspricht einem Schachbrett, mit Zahlen in der einen und Buchstaben in der anderen Dimension. Wir definieren es mit:

```
DIM AX (7,7)
```

AX ist der Name, jede Dimension hat acht Elemente, insgesamt kann das Feld 64 Werte enthalten.

Ein dreidimensionales Feld entspricht einem Quader, oder bei gleicher Elementenzahl pro Dimension (Seite) einem Würfel. Dieses wird dimensioniert mit

```
DIM BY (125,6,2)
```

Die Anzahl der Dimensionen wird nur begrenzt durch den verfügbaren Speicherplatz. Wieviel Bytes pro Feld gebraucht werden, entnehmen Sie bitte der Erklärung bei der Darstellung der Feld-Variablen (Texteinschub Nr. 11).

Ein Feld, das wie bisher gezeigt dimensioniert wird, enthält Gleitkomma-Zahlen.

Ein Feld mit ganzen Zahlen wird durch das Zeichen % nach dem Namen gekennzeichnet, also:

```
DIM CZ%(.,.,.)
```

Ein Feld mit Zeichenketten dagegen hat nach dem Namen das übliche Zeichen $, also:

```
DIM DT$(..,..,..)
```

»Wozu brauche ich Felder, wenn ich auch normale Variable verwenden kann?«, werden Sie vielleicht noch fragen.

Felder haben den großen Vorteil, daß immer dann, wenn viele Variable in einem Programm vorkommen, die alle einen gewissen Zusammenhang haben, viel Speicherplatz gespart werden kann.

Eine normale Variable braucht 7 Byte, eine Feld-Variable nur 5 oder bei ganzen Zahlen sogar nur 2 Byte. Zugegeben, vorher steht noch ein längerer Kopf, aber halt nur einmal. Und das zahlt sich bei vielen Variablen sehr rasch aus.

Und schließlich muß ich noch darauf hinweisen, daß die »Hausnummern« oder Indizes der Elemente innerhalb eines Programms durch mathematische Operationen verändert und manipuliert werden können. Aber das ist natürlich höhere Programmierkunst und geht über diese kurze Einführung hinaus.

Texteinschub Nr. 11
Darstellung der Felder-(Array)-Variablen im Speicher

Die Felder-Variablen kommen in drei Arten vor:
- als ganze Zahlen,
- als Gleitkomma-Zahlen,
- als Zeichenketten.

Sie sind in dem Texteinschub Nr. 10 »Felder in Basic« kurz beschrieben.

Wir wollen sie uns hier mit den Methoden anschauen, welche ich für den C 64 und für den VC 20 in dem Texteinschub Nr. 9 »Darstellung der normalen Variablen im Speicher« beschrieben habe.

Beim C 64 ist allerdings ein Zusatz dabei. Sie müssen, am besten gleich am Anfang, noch eingeben:

```
POKE 44,4:NEW
```

Ein eventuell auftretender SYNTAX ERROR soll uns nicht weiter stören.

Wenn Sie also das jeweilige Kochrezept ausgeführt und damit den Bildschirm- und den Variablenspeicher auf dieselbe Adresse gelegt haben, können wir anfangen.

Gleitkomma-Feld

Geben Sie direkt ein:

```
DIM AB(1,2,3)
```

Wir dimensionieren also ein Feld mit dem Namen AB, es hat drei Dimensionen, die erste Dimension hat zwei (0,1) Werte, die zweite hat drei und die dritte hat vier Werte. Sobald Sie die RETURN-Taste drücken, erscheint das Feld auf dem Bildschirm. Wir sehen folgende Zeichen:
A, B, invertiertes C, @ c @ d @ c @ b plus 120 Klammeraffen @.

Die ersten zwei Stellen sind der Name des Feldes in der Darstellung für Gleitkomma-Variable, wie in der letzten Folge beschrieben wurde. Die dritte und vierte Stelle geben im Bildschirmcode als Low- und High-Byte die Länge des Feldes an (das inverse C = 131, das @ = 0, bitte nachzählen). Die fünfte Stelle zeigt die Anzahl der Dimensionen (c = 3) an. Ab der sechsten Stelle stehen die Anzahl der Elemente der Dimension (diesmal als High- und Low-Byte) und zwar beginnend mit der letzten Dimension. In unserem Falle ist das also in Stelle 6 und 7 ein @ und d (0 - 3 =4=d), Stelle 8 und 9 sind dasselbe für die zweite Dimension und schließlich Stelle 10 und 11 für die erste Dimension (0 - 1=2=b). Danach folgen entsprechend der Anzahl der dimensionierten Elemente (2*3*4=24) 5 Byte pro Element (24*5=120), die vorerst auf 0 = @ stehen, die aber mit den Werten der Elemente aufgefüllt werden.

Dieses Auffüllen wollen wir nachvollziehen. Geben Sie bitte direkt ein:

```
AB(0,0,0)=5
```

Wir weisen damit dem allerersten Element des Feldes den Wert 5 zu.

In der oberen Darstellung des Feldes AB ändern sich dadurch Byte 12 und 13. Das neu erschienene inverse C und die Leerstelle mit den drei nachfolgenden @ ist die Gleitkomma-Darstellung (Mantisse und Exponent) der Zahl 5. Auf diese Darstellung werde ich später im Verlauf dieses Kurses bei der Besprechung der Speicherzelle 97 noch genauer eingehen.

Wenn wir jetzt (durch Überschreiben der vorigen Anweisung) zusätzlich noch eingeben:

```
AB(1,0,0)=6
```

erreichen wir eine entsprechende Änderung der Bytes 17 und 18, also des zweiten Elements des Feldes.

In Bild 9 sind die Stellen eines Gleitkomma-Feldes grafisch dargestellt.

Ganzzahliges Feld

Im Vergleich zu dem Gleitkomma-Feld dimensionieren wir als nächstes ein ganzzahliges Feld:

```
DIM AB%(1,2,3)
```

Jetzt erscheint auf dem Bildschirm gleich anschließend an das erste Feld eine neue Darstellung: invertiertes A, invertiertes B, ;, @, c@, d, @, c, @b plus 48 Klammeraffen @.

Die ersten 11 Byte haben dieselbe Bedeutung wie beim Gleitkomma-Feld, aber nur deswegen, weil wir dieselben drei Dimensionen mit identischer Elementenzahl dimensioniert haben. Bei mehr Dimensionen wäre dieser Kopf natürlich länger. Die inverse Darstellung des Feldnamens signalisiert ein ganzzahliges Feld. Die dritte Stelle zeigt das »;« - im Bildschirmcode ist das die 59. In der Tat ist das Feld nur 59 Byte lang, also wesentlich weniger als das Gleitkomma-Feld. Die 2*3*4=24 Elemente benötigen in der Ganzzahl-Darstellung nur je 2 Byte (24*2=48+11=59). Womit bewiesen ist, daß eine Ganzzahl-Darstellung mit dem Zeichen % erheblich Speicherplatz spart - allerdings nur bei Feldern!

Jetzt wollen wir noch den Inhalt des Feldes füllen, so wie vorher mit:

```
AB%(0,0,0)=5
```

... und prompt ändert sich Byte Nummer 13 in ein e (e = 5).

Eine Eingabe für das zweite Element:

```
AB%(1,0,0)=6
```

verändert das 15. Byte in ein f.

In Bild 10 ist der Inhalt eines Ganzzahl-Feldes grafisch dargestellt.

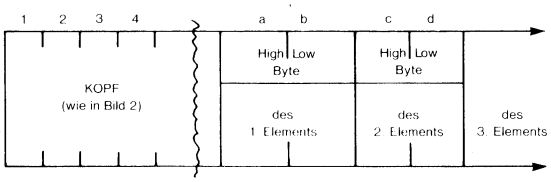


Bild 10. Ganzzahliges Feld

Felder mit Zeichenketten

Die Dimensionierung eines Feldes mit Zeichenketten sieht so aus:

```
DIM AB$(1,2,3)
```

Auf dem Bildschirm erscheint jetzt ein Feld:

Auch hier zeigen die ersten elf Stellen dieselbe Information wie bei den anderen Feldern. Zur Kennzeichnung des Zeichenketten-Feldes ist das zweite Zeichen des Feldnamens invers dargestellt. Zeichen 3 und 4 geben wieder die Länge des Feldes an. Das S hat den Bildschirmcode 83. (Vorsicht! Da wir im Groß-/Kleinbuchstaben-Modus sind, müssen wir die jeweils rechte Seite der Spalten in der Code-Tabelle nehmen). Die Länge 83 minus 11 Kopfstellen ergibt 72 Byte, geteilt durch 24 (2*3*4 = 24 Elemente) erhalten wir 3 Byte zur Darstellung eines Elements.

Das erste Byte gibt die Länge der Zeichenkette an, das zweite und dritte Byte (Low-/High-Byte) die Adresse, ab der die Zeichenkette im vierten Block gespeichert ist.

Die Methode ist also dieselbe wie bei den »normalen« Zeichenketten-Variablen. Das wollen wir uns auch noch ansehen. Geben Sie direkt ein:

```
AB$(0,0,0)="AAAAAA"
```

In der Darstellung des Feldes ändern sich dadurch die Stellen 12, 13 und 14 und wir sehen
— beim C 64:
— beim VC 20:
Im Bildschirm steht dafür:
— C 64: 6 250 159 das heißt 6 Zeichen,
ab Adresse 250+159*256=40959
— VC 20: 6 250 29 das heißt 6 Zeichen
ab Adresse 250+29*256=7674

Jetzt weisen wir dem letzten Element auch noch eine Zeichenkette zu:

```
AB$(1,2,3)="BB"
```

Die letzten drei Stellen des Feldes ändern sich ebenfalls, wobei die erste mit dem b eine Zeichenkettenlänge von 2 angibt, dementsprechend muß die Anfangsadresse um 2 niedriger sein als die vorher definierte Kette: Das Low-Byte 250 - 2 = 248, in der Codetabelle finden wir dafür das, was auch im Feld steht. Das High-Byte bleibt unverändert.

Bild 11 zeigt die grafische Darstellung des Zeichenketten-Feldes.

Als letztes zeige ich Ihnen noch die im vierten Block gespeicherten Zeichenketten. Wir drucken einfach den CHR$-Wert der in den betreffenden Speicherzellen stehenden Codezahlen aus mit:

— VC 20:

```
FOR I=248 TO 255:PRINT CHR$(PEEK(29*256+I));:NEXT
```

— C 64:

```
FOR I=248 TO 255:PRINTCHR$(PEEK(159*256+I));:NEXT
```

... und wir erhalten die beiden Zeichenketten in umgekehrter Reihenfolge, also vom Speicherende her eingespeichert. Interessant ist, daß sich vor die Felder - wenn Sie sie noch auf dem Bildschirm hatten - die neu definierte Gleitkomma-Variable I@ geschoben hat. Auch das ist eine Demonstration des Speicherverfahrens der Variablen, genauso wie ich es Ihnen in der letzten Folge erklärt habe.

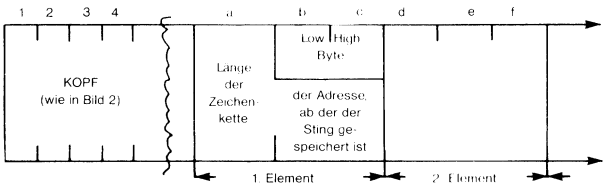


Bild 11. Zeichenketten-Feld

in der Tat die jeweilige Zeilennummer an.

Die Basic-Befehle GOTO, GOSUB-RETURN, FOR-NEXT, END, STOP, CONT und die Betätigung der STOP-Taste während eines Programmlaufes verwenden alle den Inhalt dieser Speicherzellen, um entweder zu der laufenden Zeile zurückzufinden oder um die Unterbrechung mit BREAK IN... anzuzeigen. Auch die meisten Fehlermeldungen verwenden diese Zellen.

In vielen Basic-Erweiterungen und Programmierhilfen wird ein Befehl TRACE oder STEP angeboten, welcher ein schrittweises Abarbeiten eines Programms bei gleichzeitiger Anzeige der gerade aktiven Zeilennummer erlaubt. Dieses TRACE verwendet natürlich auch den Inhalt der Zellen 57 und 58.

Schließlich sei noch erwähnt, daß im direkten Modus, also bei direkt eingetippten Aktionen des Computers ohne Programmzeilen, in der Zelle 58 immer die Zahl 255 steht. Diejenigen Basic-Befehle, welche im direkten Modus nicht erlaubt sind (INPUT, GET, DEF), prüfen in Zelle 58, ob sie im direkten Modus oder während eines Programmlaufes aufgetreten sind.

Adresse 59 und 60 ($3B und $3C)

Zeilennummer der letzten Programmunterbrechung

Immer dann, wenn ein Programmablauf durch die Befehle END oder STOP oder aber mit der STOP-Taste abgebrochen wird, wird die Nummer der gerade ausgeführten Programmzeile nach 59 und 60 gebracht und bleibt dort solange, bis eine neue Unterbrechung erfolgt.

Das läßt sich am besten mit der STOP-Taste und nachfolgendem CONT zeigen. Nehmen Sie bitte dazu das kleine Demo-Programm der Zellen 57 und 58 und ändern Sie alle PEEK-Adressen in 59 und 60 um. Fügen Sie außerdem noch eine Zeile 50 hinzu:

```
50 GOTO 10
```

Den dadurch erzeugten kontinuierlichen Lauf des Programms bremsen Sie dann mit der STOP-Taste und lassen ihn danach mit CONT weiterlaufen.

Auf der rechten Seite erscheint jetzt die Zeilennummer, bei der das Programm vorher unterbrochen worden ist.

Adresse 61 und 62 ($3D und $3E)

Zeiger auf die Adresse, ab welcher der Text der laufenden Basic-Zeile gespeichert ist.

Die Abarbeitung der einzelnen Basic-Zeilen während eines Programmlaufs wird von einem kleinen Maschinencode-Programm, welches in den Speicherzellen 115 bis 138 steht (wir kommen noch dahin), gesteuert. In den Zellen 122 und 123 enthält es die Adresse des letzten Bytes des gerade ausgeführten Basic-Befehls.

Sobald eine neue Basic-Zeile verarbeitet wird, holt das Betriebssystem diese Adresse aus 122 und 123 und speichert sie in den hier zur Diskussion stehenden Speicherzellen 61 und 62 ab, wie üblich als Low-/High-Byte.

Dasselbe geschieht bei jedem Befehl END, STOP, bei Fehlern mit dem Befehl INPUT und durch das Drücken der STOP-Taste. Der Befehl CONT hingegen schaut in 61 und 62 nach und bringt die darin befindliche Adresse zurück in die Speicherzellen 122 und 123 zur Fortsetzung des Programms. Wenn aber in Zelle 62 inzwischen eine 0 steht - und das geschieht bei einem LOAD-Befehl, durch Programm-Abbruch mit Fehlermeldung und durch Eingabe neuer Basic-Zeilen beziehungsweise deren Veränderungen mit abschließender RETURN-Taste - dann wird der CONT-Befehl nicht ausgeführt.

Zur besseren Erklärung dieser in 61 und 62 als Zeiger stehenden Adresse einer Basic-Zeile möchte ich Sie an den Texteinschub Nr. 7 erinnern, in dem ich den Basic-Programmspeicher »sichtbar« gemacht habe, um die Wirkung der Verschiebung des Zeigers in den Zellen 43 und 44 zu demonstrieren.

Wir nehmen dazu bitte noch einmal das kleine Demo-Programm für die Adressen 57 und 58 oben her und ersetzen die PEEK-Werte durch 61 und 62. Das Ausdrucken des Inhalts von 61 und 62 legen wir aber an den Anfang jeder Zeile. Das Programm sieht dann so aus:

```
10 PRINT PEEK(61)+256*PEEK
(62),"ZEILE 10"
20 PRINT PEEK(61)+256*PEEK
(62),:A=3:PRINT A
30 PRINT PEEK(61)+256*PEEK
(62),:B=5:PRINT B
40 PRINT PEEK(61)+256*PEEK
(62),A*B
```

Nach RUN erhalten wir jetzt auf der linken Seite Zahlen, die den jeweiligen Basic-Speicher angeben, ab dem diese Zeile gespeichert ist. Wenn Sie ab diesen Adressen mit der gerade erwähnten Methode aus Texteinschub Nr. 7 nachschauen, finden Sie genau die Zeilen des kleinen Demo-Programms wieder.

Zur Anwendung dieses Zeigers kann ich wenig sagen. Ihn durch POKE zu verändern, geht in Basic nicht, weil das Betriebssystem die richtigen Werte immer neu eingibt. Man kann ihn allerdings abfragen, wenn man sich für die Speicheradressen der Basic-Zeilen interessiert. Die einzige Anwendung dafür kenne ich von S. Leemon, welche bei den Adressen 65 und 66 eingesetzt wird.

Adresse 63 und 64 ($3F und $40)

Zeilennummer eines gerade laufenden DATA-Befehls

Diese Speicherzellen enthalten die Nummer der Basic-Zeile, in der gerade ein DATA-Befehl mit READ gelesen wird. Sobald in einer DATA-Zeile ein Fehler gefunden wird, kommt diese Zeilennummer aus 63 und 64 in die Speicherzellen 57 und 58, um in der Fehlermeldung die fehlerhafte DATA-Zeile und nicht die laufende READ-Zeile anzuzeigen. Auf diese Weise werden Syntax-Fehler in einer DATA-Zeile angezeigt. Um andere Fehler, wie zum Beispiel ein fehlendes Komma zwischen zwei DATA-Angaben anzuzeigen, können die Speicherzellen 63 und 64 eingesetzt werden.

In dem folgenden Programm wird in Zeile 20 geprüft, ob die DATA-Angaben größer als 255 sind. Da bei einem fehlenden Komma die beiden Zahlen als eine Zahl gelesen werden, wird dieser Fall erkannt und mit einem F versehen die Nummer der DATA-Zeile ausgedruckt, in der das Komma fehlt.

```
10 FOR X=1 TO 10:READ
A:PRINT A
20 IF A>255 THEN PRINT
"F" PEEK(63) + 256*PEEK
(64)
30 NEXT X
40 DATA 10,20,30
50 DATA 40,50,60
60 DATA 70,80,90,100
```

Sie können jetzt in den DATA-Zeilen Kommafehler einbauen, die vom Programm angezeigt werden. Ein anderer häufiger Fehler, nämlich ein Komma am Ende einer DATA-Zeile, kann damit leider nicht erkannt werden. Aber vielleicht fällt Ihnen eine Prüfformel dazu ein.

Adresse 65 und 66 ($41 und $42)

Zeiger auf die Adresse, ab der die laufende DATA-Angabe gespeichert ist

Diese Speicherzellen enthalten in der Low-/High-Byte-Darstellung die Adresse im Basic-Programmspeicher, ab welcher der READ-Befehl nach der nächsten DATA-Zeile sucht.

Zu Beginn eines Programms steht in 65 und 66 als Adresse der Beginn des Basic-Speichers, also derselbe Wert wie in den Speicherzellen 43 und 44. Der Befehl RESTORE setzt den Zeiger immer auf diesen Anfangswert zurück. Ein Demo-Programm zeigt uns das an (die Kommata sind wichtig für das Format der Darstellung auf dem Bildschirm!):

```
10 PRINT, PEEK(65)+256*
PEEK(66)
20 FOR X=1 TO 10:READ A
30 PRINT A,PEEK(65)+
256*PEEK(66)
40 NEXT X
50 DATA 10,20,30,40,50,60,
70,80,90,100
60 RESTORE
70 PRINT,PEEK(65)+256*PEEK
(66)
```

Durch Verändern dieses Zeigers in 65 und 66 kann die Reihenfolge, mit der DATA-Angaben gelesen werden, verändert werden, allerdings nur zeilenweise.

Wir brauchen dazu die oben beschriebenen Speicherzellen 61 und 62, deren jeweiligen Inhalt wir ja mit PEEK abfragen können. Wenn wir das vor jeder DATA-Zeile machen und diesen Wert einer Variablen zuweisen, haben wir die Adresse gespeichert, hinter welcher die DATA-Zeile kommt. Durch POKEn dieser Adressen in die Speicherzellen 65 und 66 vor einem READ-Befehl, wird diesem READ die nächste DATA-Zeile vorgegeben und wir können so die Reihenfolge der DATA-Zeilen ändern.

```
10 A1=PEEK(61)+PEEK(62)*256
20 DATA DAS IST DIE 1. ZEILE
30 A2=PEEK(61)+PEEK(62)*256
40 DATA DAS IST DIE 2.ZEILE
50 A3=PEEK(61)+PEEK(62)*256
60 DATA DAS IST DIE 3.ZEILE
70 POKE 65,A3 AND 255:
 POKE 66,A3/256:
 READ A$:PRINT A$
 80 POKE 65,A1 AND 255:
 POKE 66,A1/256:
READ A$:PRINT A$
 90 POKE 65,A2 AND 255:
 POKE 66,A2/256:
 READ A$:PRINT A$
```

Mit den Zeilen 70 bis 90 werden für jede DATA-Zeile eigene READ-Anweisungen gegeben. Welche DATA-Zeile gelesen werden soll, wird durch die Variablen Ax und Bx (x=1,2,3) bestimmt, mit denen der Zeiger in 65 und 66 »verbogen« wird. Auf ein Detail will ich hier hinweisen:

Die Adresse 61 und 62 darf nicht mit zwei Befehlen, sondern muß mit einem Befehl ausgelesen werden, da bei einem möglichen Page-Wechsel zwischen den zwei Befehlen der Zeiger nicht verbogen, sondern abgeknickt wird.

Was passiert in der ersten Zeile des Demo-Programms?

```
10 A1=PEEK(61):B1=PEEK(62)
```

Mit »A1=PEEK(61)« wird der Variablen A1 der Wert des Low-Bytes des Zeigers 61 und 62 zugewiesen. Dieser zeigt am Anfang einer Zeile auf das Null-Byte vor der Linkadresse (hier 2048), so daß A1 den Wert (2048 AND 255)=0 erhält. Mit »B1=PEEK(62)« wird der Variablen B1 der Wert des High-Bytes des Zeigers 61 und 62 zugewiesen. Dieser zeigt aber inzwischen auf das Trennzeichen (»:«) zwischen den beiden Befehlen (hier 2061), so daß B1 den Wert (INT(2061/256))=8 erhält. Als Zeiger auf das aktuelle DATA-Element erhalten wir die erwartete Adresse (A1+B1*256)=2048.

Was aber, wenn Zeilenanfang und Trennzeichen nicht in derselben Page liegen? Dazu setzen Sie bitte den Basic-Anfang um eine Stelle zurück:

```
POKE43,0:POKE 2047,0:NEW
```

Die Zeiger auf den Zeilenanfang und das Trennzeichen werden dadurch ja ebenfalls verändert, so daß A1 jetzt den Wert (2047 AND 255)=255 und B1 den Wert (INT(2060/256))=8 erhält. Als Zeiger auf das aktuelle DATA-Element erhalten wir nun die völlig unbrauchbare Adresse (A1+B1*256)=2303.

Adresse 67 und 68 ($43 und $44)

Zeiger auf die Adresse, aus welcher die Befehle INPUT, GET und READ die Zeichen/Zahlen holen

INPUT und GET verlangen Angaben, die per Tastatur eingegeben werden. Tastatur-Eingaben im direkten Modus, also wenn kein Programm läuft, werden im Eingabe-Pufferspeicher des Editors (der Teil des Betriebssystems, welcher für die Zeilendarstellung auf dem Bildschirm verantwortlich ist) ab Speicherzelle 512 bis 600 zwischengespeichert.

Der Zeiger in 67 und 68 zeigt auf die jeweilige Adresse in diesem Eingabe-Pufferspeicher. Bei READ ist 67 und 68 identisch mit 65 und 66. Der Inhalt dieser Speicherzellen kann mit PEEK ausgelesen werden.

Adresse 69 und 70 ($45 und $46)

Name der gerade aufgerufenen Basic-Variablen

Wenn beim Ablauf eines Programms eine Variable auftaucht, muß ihr derzeitiger Wert im Variablen-Speicher gesucht werden. Während dieses Suchvorgangs wird der Name der Variablen in 69 und 70 zwischengespeichert. Die Form der Zwischenspeicherung ist dieselbe 2-Byte-Darstellung wie im Variablenspeicher, beschrieben bei der Behandlung der Speicherzellen 45 und 46.

Adresse 71 und 72 ($47 und $48)

Zeiger auf die Adresse des Wertes der gerade aufgerufenen Basic-Variablen

Ähnlich wie bei 69 und 70 wird hier während des Anrufes einer Variablen durch ein Programm ein Wert zwischengespeichert, diesmal aber nicht der Name der Variablen, sondern der 2-Byte-Wert, welcher direkt hinter dem Variablennamen steht. Nähere Einzelheiten sind im Text der Speicherzellen 45 und 46 beschrieben.

Davon ausgenommen sind selbstdefinierte Funktionen. Wie im Texteinschub Nr. 12 »Darstellung der Variablen einer selbstdefinierten Funktion« gezeigt ist, erscheinen diese ebenfalls im Variablenspeicher in einer Darstellung, welche den normalen Variablen sehr ähnlich ist.

Damit nun eine normale oder Feld-Variable denselben Namen haben kann wie eine Funktion, wird die oben genannte Zwischenspeicherung in 69 und 70 bei Funktionen unterdrückt.

Adresse 73 und 74 ($49 und $4A)

Zwischenspeicher für Variable einer FOR-NEXT-Schleife und für diverse Basic-Befehle

Die Adresse einer Schleifenvariablen wird zuerst hier gespeichert, bevor sie auf den Stapelspeicher ab Speicherzelle 256 ($100) gebracht wird. Die Funktion und Arbeitsweise des Stapelspeichers werden wir bei diesen Adressen behandeln. Etliche Basic-Befehle, wie LIST, WAIT, GET, INPUT, OPEN, CLOSE und andere, verwenden die Speicherzellen 73 und 74 für Zwischenspeicherungen. Diese Adressen sind für den Basic-Programmierer daher nicht verwendbar.

Adresse 75 und 76 ($4B und $4C)

Zwischenspeicher für Zeiger bei READ und mathematischen Operationen

Während der Auswertung eines mathematischen Ausdrucks durch die Routine FRMEVL des Basic-Übersetzers, wird der Platz des betroffenen mathematischen Operators in einer Tabelle, hier in 75 und 76, zwischengespeichert. Dieser Platz wird dabei als Abstand zum Beginn der Tabelle dargestellt. Außerdem verwendet der READ-Befehl diese Adressen als Zwischenspeicher für einen Programmzeiger. Die Speicherzeilen 75 und 76 sind in Basic nicht verwendbar.

Adresse 77 ($4D)

Hilfsspeicher für Vergleichs-Operationen

Die bei 75 und 76 schon erwähnte Auswertungs-Routine FRMEVL erzeugt in der Speicherzelle 77 einen Wert, der angibt, ob es sich bei einer Vergleichsoperation um den Fall »kleiner als« (<), »gleich wie« (=) oder »größer als« (>) handelt. Diese Speicherzelle ist nur im Maschinencode erreichbar.

Adresse 78 und 79 ($4E und $4F)

Zeiger auf Adresse, ab welcher der Wert der Variablen einer selbstdefinierten Funktion gespeichert ist

Basic erlaubt es bekanntlich, mit dem Befehl DEF selbst erfundene Funktionen zu definieren, welche die Form FN gefolgt von einem Variablennamen haben, zum Beispiel

```
DEF FNAA(X).
```

Im Texteinschub Nr. 12 »Darstellung von Variablen selbstdefinierter Funktionen« wird gezeigt beziehungsweise sichtbar gemacht, wie derartige Funktionen und ihre Variablen gespeichert werden. Während der Definition einer Funktion steht in 78 und 79 die Adresse, ab welcher die Funktion und der Wert ihrer Variablen gespeichert ist. Der Inhalt dieser Adressen ist identisch mit den Zeichen hinter dem Namen der Funktion (1. Gruppe im nebenstehenden Beispiel).

Nach der Ausführung der Funktion steht in 78 und 79 allerdings die Adresse, ab welche der Zahlenwert der Funktion selbst gespeichert ist. Er ist identisch mit den Zeichen der 2. Gruppe.

Diesen Zusammenhang können Sie überprüfen, indem Sie im Programm des Texteinschubes folgende Zeilen hinzufügen:

```
25 PRINT PEEK(78)+256*
PEEK(79)
35 PRINT PEEK(78)+256*
PEEK(79)
```

Nach RUN erhalten Sie zwei Adressen, die Sie mit direkter Eingabe abfragen:

```
FOR I=0 TO 4:PRINT PEEK
(1.Adresse+I);:NEXT I:
FOR J=0 TO 4:PRINT PEEK
(2.Adresse+J);:NEXT J
```

Sie werden sehen, daß der Inhalt der beiden Adressen genau den Werten der Zeichen 3 bis 7 der beiden Gruppen entspricht, allerdings im Bildschirmcode.

Adresse 80 bis 82 ($50 bis $52)

Zeiger auf einen provisorischen Speicherplatz einer Zeichenkette, die gerade bearbeitet wird

Die Teilprogramme (von Programmierern »Routinen« genannt) des Basic-Übersetzers im ROM des Computers, welche Zeichenketten (Strings) behandeln, verwenden die ersten beiden Bytes dieser drei Speicherzellen, nämlich 80 und 81, um in Low-/High-Byte-Darstellung diejenige Speicheradresse anzugeben, ab der die Zeichenkette im Programmspeicher zu finden ist.

Das dritte Byte (82) enthält die Länge der Zeichenkette. Wegen der provisorischen Natur dieses Zeigers ist er für Basic-Programme nicht geeignet.

Adresse 83 ($53)

Flagge für die Garbage Collection

In dieser Speicherzelle steht während der sogenannten Garbage Collection (Müllabfuhr) eine Zahl, die angibt, ob die Variable der zur Überprüfung anstehenden Zeichenkette eine Länge von 3 oder 7 Byte hat.

Der Vorgang der Garbage Collection ist von B. Schneider, 64'er-Ausgabe 1/85, ausführlich beschrieben worden. Angaben über die Bedeutung der Variablen einer Zeichenkette finden Sie in den Texteinschüben Nr. 9 und Nr. 11.

Adresse 84 bis 86 ($54 bis $56)

Sprungbefehl auf die Adressen der Basic-Funktionen

Jede Basic-Funktion, wie zum Beispiel SGN, INT, ABS, USR und so weiter, wird durch ein spezielles Teilprogramm (Routine) des Basic-Übersetzers ausgeführt. Die Anfangsadresse jeder dieser Routinen sind in einer Tabelle im ROM fest eingespeichert. Im VC 20 steht diese Tabelle von 49234 bis 49279 ($C052 bis $C07F), im C 64 von 41042 bis 41087 ($A052 bis $A07F).

In der Speicherzelle 84 steht der Sprungbefehl JMP in Maschinencode, dargestellt

Texteinschub Nr. 12
Darstellung der Variablen einer selbstdefinierten Funktion

Ich habe Ihnen gezeigt, wie im Programmspeicher abgelegte normale Variablen und Felder-Variablen sichtbar gemacht werden können. Damit konnten wir den Aufbau und die Darstellung der einzelnen Variablenarten studieren.

Heute will ich einen weiteren Variablentyp vorstellen, nämlich den der selbstdefinierten Funktionen.

Sie erinnern sich vielleicht, mit dem Basic-Befehl »DEF FN (Name)(Variable)« können wir komplizierte Funktionen selbst erfinden, definieren und später als »FN (Name)(Variable)« weiter verarbeiten. Diesen Typ wollen wir uns anschauen, wie er im Speicher steht.

Im Prinzip verwenden wir dieselben Methoden zur Sichtbarmachung wie die letzten Male.

Aber ein Unterschied kommt noch dazu. Der Befehl DEF kann leider nicht direkt eingegeben werden, sondern muß immer als Teil einer Programmzeile mit einer Zeilennummer versehen sein.

Deshalb schreiben wir zuerst ein kleines Programm zur Definition der Funktion plus Variable, bevor wir den Variablenspeicher mit dem Bildschirmspeicher zusammenlegen:

```
10 DEF FNAA(X)=3*SIN(X)+COS(X)
20 X=5
30 PRINT FNAA(X)
```

Die Funktion hat also den Namen »AA«. Bevor wir weitermachen, überprüfen Sie bitte mit RUN, ob alles stimmt. Nun wird der Speicher verschoben.

Für den C 64 gilt:

1. POKE 46,4:POKE 48,4
2. Bildschirm löschen mit CLR-Taste
3. Cursor auf die Mitte fahren
4. LIST (es erscheint das Programm)
5. auf den 2. Zeichensatz umschalten (mit C= und SHIFT-Taste)
6. RUN

Für den VC 20 (ohne Erweiterung) gilt:

Nur den Bildschirm auf 4096 zu verschieben, wie das letzte Mal, geht diesmal nicht, da wir ja für DEF ein kleines Programm schreiben müssen.

Also legen wir Bild- und Variablenspeicher ab Adresse 5120 (5120/256=20).

1. POKE 46,20:CLR
2. POKE 648,20
3. STOP/RESTORE-Tasten, bis Cursor wieder da ist
4. Bildschirm löschen mit CLR-Taste
5. die ersten vier bis sechs Zeilen mit SPACE-Taste überfahren
6. Cursor ein paar Zeilen nach unten
7. LIST (es erscheint das Programm)
8. mit Commodore- und SHIFT-Taste auf 2. Zeichensatz umschalten
9. RUN

Wir sehen jetzt oben zwei Gruppen mit je sieben Zeichen, wie üblich.

Die erste Gruppe stellt die Funktion FNAA(x) dar. Sie ist gekennzeichnet durch das invertierte erste Zeichen des Namens, während das zweite Zeichen normal erscheint.

Das dritte und vierte Zeichen gibt in Low-/High-Byte-Darstellung (im Bildschirmcode) die Adresse an, ab der die Funktion FNAA(x) im Programmspeicher abgelegt ist. Mit `PEEK (3. Zeichen)+256*PEEK (4. Zeichen)` kann das abgefragt werden.

Das fünfte und sechste Zeichen nennt die Adresse, an welcher der Zahlenwert der Funktions-Variablen X anfängt. Das siebente Zeichen schließlich ist das erste Zeichen der Funktion selbst (in unserem Beispiel die 3).

Die zweite Gruppe beschreibt die Variable X der Funktion. Die normale Darstellung der beiden ersten Zeichen, die den Namen darstellen, gibt uns an, daß es sich um eine Gleitkomma-Variable handelt, deren Wert als Mantisse und Exponent dargestellt ist.

Der Aufbau einer Funktion ist in Bild 12 zusammengefaßt:

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>Erstes</td><td>Zweites</td><td>Low-</td><td>High-</td><td>Low-</td><td>High-</td><td></td></tr>
<tr><td colspan="2">Zeichen des Funktionsnamens</td><td colspan="2" rowspan="2">Byte der Adresse, ab der die Funktion abgespeichert ist</td><td colspan="2" rowspan="2">Byte der Adresse, ab dem der jeweilige Wert der Funktionsvariablen X abgespeichert ist</td><td rowspan="2">1. Zeichen der Funktion</td></tr>
<tr><td>ASCII-Wert + 128</td><td>ASCII-Wert</td></tr>
</table>

Bild 12. Selbstdefinierte Funktion

durch die Zahl 75 ($4C). In den beiden anderen Zellen 85 und 86 steht dann in Low-/High-Byte-Darstellung die jeweilige Adresse in der Tabelle, welche der vom Programm gerade gebrauchten Basic-Funktion entspricht. Dieser gesamte Befehl JMP plus Adresse entspricht in Basic der GOSUB-Zeilennummer.

Ein Beispiel soll das verdeutlichen. Geben Sie direkt ein:

```
PRINT PEEK(84);PEEK(85);
PEEK(86)
```

Wir erhalten
beim C 64: 76 13 184
beim VC 20: 76 13 216

Die erste Zahl ist genauso wie oben beschrieben. Die beiden anderen Zahlen ergeben zusammen die Adresse 47117 ($B80D) beziehungsweise 55309 ($D80D). Wenn Sie ein Buch mit ROM-Listing haben, werden Sie unter dieser Adresse die Routine für die Funktion »PEEK« finden. Das ist natürlich nicht erstaunlich, haben wir doch gerade vorher als letzten Befehl genau diese Funktion eingegeben.

Leider ist das auch die einzige Funktion, die ich Ihnen vorführen kann, denn zum Vorführen muß ich eben immer PEEKen, so daß beim besten Willen immer nur die oben angegebenen Zahlen erscheinen können.

Adresse 87 bis 96 ($57 bis $60)

Arbeitsspeicher für diverse Arithmetik-Routinen des Basic-Übersetzers

Diese zehn Speicherplätze werden von verschiedenen Teilprogrammen (Routinen), besonders bei arithmetischen Operationen, als Zwischenspeicher verschiedener Werte, Flaggen und Zeiger benutzt.

Adresse 97 bis 102 ($61 bis $66)

Gleitkomma-Akkumulator Nr.1

»Akkumulator« heißt seit der Zeit der mechanischen Rechenmaschinen eine Speicherzelle, welche bei Rechenoperationen dadurch im Mittelpunkt steht, daß laufend Daten in sie hineingeschrieben beziehungsweise aus ihr herausgelesen werden.

Normalerweise trägt diesen Namen das zentrale Rechenregister des Mikroprozessors. Leser des Assembler-Kurses kennen diesen Akkumulator inzwischen zur Genüge.

Die Speicherzellen 97 bis 102 werden deswegen ebenfalls Akkumulator genannt, weil sie bei der Verarbeitung von Gleitkommazahlen eine ähnliche zentrale Rolle spielen.

Zelle 97 enthält den Exponenten. Die Zellen 98 bis 101 enthalten die Mantisse.

Zelle 102 enthält das Vorzeichen der Gleitkommazahl. Eine 0 bedeutet ein positives, die Zahl 255 ein negatives Vorzeichen.

Mit dem Gleitkomma-Akkumulator Nr. 1 sind zwei weitere Speicherzellen eng verbunden, nämlich 104 ($68) und 112 ($70).

Ganz zum Schluß ist noch erwähnenswert, daß nach der Umwandlung einer Gleitkommazahl in eine ganze Zahl diese als Low-/High-Byte in den beiden Speicherzellen 98 und 99 steht, was für Maschinenprogramme vielleicht recht nützlich sein kann.

Adresse 103 ($67)

Zwischenspeicher beziehungsweise Zählregister

Diese Adresse wird von zwei Routinen verwendet. Der Basic-Übersetzer benutzt sie als Vorzeichenspeicher bei der Umwandlung von Zahlen aus dem ASCII-Format in Gleitkommazahlen. Das Betriebssystem verwendet diese Adresse als Zähler der Abarbeitungsschritte bei der Berechnung eines Polynoms der Form

$y=a0+a1*x+a2*x\uparrow 2+a3*x\uparrow 3+.....$

Adresse 104 ($68)

Überlauf-Speicher des Gleitkomma-Akkumulators Nr. 1

Wenn eine Zahl so groß wird, daß sie mit den zur Verfügung stehenden Stellen nicht mehr dargestellt werden kann, sprechen wir von einem »Überlauf«.

Bei Gleitkommazahlen liegt diese Überlaufgrenze bei $1{,}70141183 * 10^{38}$.

Während einer mathematischen Berechnung kann es intern im Computer vorkommen, daß ein Überlauf eintritt, der aber am Ende der Operation wieder verschwinden würde. Der Akkumulator Nr. 1 benutzt in einem derartigen Fall die Speicherzelle 104, um die verfügbare Stellenzahl um 8 Bit zu vergrößern. Für endgültige Resultate steht diese Erweiterung natürlich nicht zur Verfügung.

Dieser Vorgang tritt besonders häufig bei der Umwandlung von ganzen Zahlen oder Zeichenketten in Gleitkommazahlen auf.

Adresse 105 bis 110 ($69 bis $6E)

Gleitkomma-Akkumulator Nr. 2

Spätestens jetzt verstehen Sie, warum der Akkumulator der Speicherzellen 97 bis 102 die Nr. 1 hat. Es gibt hier noch einen zweiten Gleitkomma-Akkumulator, der ein identischer Zwilling ist. Zwei Akkumulatoren sind immer dann notwendig, wenn mathematische Operationen ablaufen, welche mehr als einen Operanden verarbeiten, wie zum Beispiel Multiplikation, Division und so weiter.

Aufgrund der Identität der beiden Akkumulatoren kann ich mir eine weitere Beschreibung ersparen.

Adresse 111 ($6F)

Flagge für Vorzeichenvergleich der Gleitkomma-Akkumulatoren Nr. 1 und Nr. 2

Wenn die Zahl in beiden Akkumulatoren gleiche Vorzeichen hat, steht in Speicherzelle 111 eine 0, bei verschiedenen Vorzeichen eine 255.

Adresse 112 ($70)

Rundungsspeicher des Gleitkomma-Akkumulators Nr. 1

Es kann vorkommen, daß die Mantisse einer Gleitkommazahl mehr Stellen hat, als mit den vier Mantissen-Bytes des Akkumulators Nr. 1 (Zelle 97 bis 102) dargestellt werden können. In diesem Fall werden die hintersten, das heißt die unwichtigsten Stellen hinter dem Komma in der Zelle 112 abgelegt. Von dort werden sie geholt, um die Genauigkeit von mathematischen Operationen zu erhöhen und auch um Endresultate abrunden zu können.

Adresse 113 und 114 ($71 und $72)

Zwischenspeicher für verschiedene Routinen

Diese Speicherzellen werden von sehr vielen Routinen des Übersetzers und des Betriebssystems, wie zum Beispiel Zeichenkettenverarbeitung, interne Uhr (TI$), Bestimmung der Größe von Feldern (Arrays) und etlichen anderen verwendet.

Adresse 115 bis 138 ($73 bis $8A)

Teilprogramm »Nächstes Zeichen eines Basic-Textes holen« (CHRGET-Routine)

Die Problematik der Übersetzung von Basic-Befehlen und Anweisungen besteht darin, daß die Übersetzungsschritte durch entsprechende Programmteile des Basic-Übersetzers im Computer fest vorprogrammiert sein müssen, was bedeutet, daß diese Programme natürlich im – nicht veränderbaren – ROM stehen.

Auf der anderen Seite verlangt aber der Übersetzungsvorgang, daß gewisse Teile dieser Programme sich laufend verändern. Als Beispiel soll der Zeiger herhalten, der angibt, in welcher Speicherzelle das nächste zu bearbeitende Zeichen steht. Dieser Zeiger und die zusammengehörigen Programmschritte dürfen natürlich nicht im ROM stehen, denn da sind sie ja nicht änderbar.

Dieser Konflikt wird dadurch gelöst, daß dieses »variable« Teilprogramm des Übersetzers zwar im ROM steht (im C 64 ab 58274 oder $E3A2, im VC 20 ab 58247 oder $E387), von wo es aber direkt nach dem Einschalten des Computers in das RAM, und zwar in die Speicherzellen 115 bis 138, umgeladen wird.

Dieses Teilprogramm, welches die Zeichen zur Übersetzung herbeiholt und deswegen »Character-Get« oder kurz CHARGET-Routine genannt wird, ist wegen seiner Veränderbarkeit natürlich ein beliebtes Objekt aller möglichen Manipulationen. Es ist deshalb im Assembler-Kurs, Teil 5, im 64'er, Ausgabe 1/85, im Detail beschrieben worden, allerdings mit Schwerpunkt auf Assembler-Maschinensprache.

Für Basic-Programmierer möchte ich hier deshalb eine kurze Beschreibung der CHARGET-Routine einfügen.

Die Routine beginnt mit einem Sprung auf den oben schon erwähnten Zeiger in Adresse 122 und 123, welcher seinerseits auf die Adresse zeigt, in welcher das nächste zu übersetzende Zeichen steht. Das Zeichen wird entsprechend dem Hinweis des Zeigers geholt, in den Akkumulator des Mikroprozessors geladen und dort verschiedenen Prüfungen unterzogen. Ist das Zeichen ein Gänsefuß, erkennt das Programm, wie es das nächste Zeichen interpretieren und behandeln muß. Ein Doppelpunkt leitet einen neuen Befehl ein, eine Leerstelle wird unterdrückt und so weiter.

Mit dem Befehl

```
PRINT PEEK(122)+256*
PEEK(123)
```

können wir innerhalb eines Programms ausdrucken, wohin der Zeiger nach dem letzten Basic-Zeichen deutet. Eine Überprüfung mit den Methoden, die ich bei der Besprechung der Speicherzellen 43 bis 56 genannt habe, zeigt Ihnen den Zusammenhang.

Normalerweise wird der Zeiger in 122 und 123 nach jedem Zeichen um 1 erhöht, da ja die Zeichen einer Basic-Zeile hintereinander im Speicher stehen. Ein GOTO- oder GOSUB-Befehl kann diese Folge natürlich unterbrechen, ebenso wie eine willkürliche Änderung durch einen Eingriff von außen.

Ein derartiger Eingriff, auch »wedge« (Keil) genannt, öffnet natürlich Tür und Tor für Programmiertricks, insbesondere für Einbau von neuen, selbsterfundenen Befehlen. Man kann entweder den allerersten Sprungbefehl auf den Zeiger so umlenken, daß er auf ein eigenes Maschinenprogramm springt, oder man kann den Zeiger selbst »verbiegen«, so daß er auf eine andere Adresse und damit auf ein anderes Zeichen zeigt. Es gibt dafür viele Möglichkeiten, die aber alle nur in Maschinencode funktionieren. Theoretisch können wir natür-

Texteinschub Nr. 13
Wie zufällig sind Zufallszahlen?

Der Befehl RND(X) ergibt eine Zufallszahl zwischen 0 und 1 – so steht es im Commodore-Handbuch.

Eine Zufallszahl ist definitionsgemäß rein dem Zufall überlassen. Ihr Wert kann nicht vorhergesehen werden. Wie kann aber ein Computer, in dem alle Vorgehensweisen und Arbeitsschritte fest vorprogrammiert sind, eine zufällige Zahl erzeugen? Die Commodore-Computer machen das so:

Der Befehl RND nimmt eine bestimmte Ausgangszahl (auf die ich noch näher eingehen werde), auf englisch »seed« = Samen genannt, multipliziert sie mit 11879546.4 und zählt 3.92767778 * 10^8 dazu. Die 5 Byte der resultierenden Gleitkommazahl werden miteinander vertauscht und in einen positiven Bruch umgewandelt. Diese Manipulation ergibt die »Zufallszahl«, die als neuer »Samen« in den Speicherzellen 139 bis 143 gespeichert wird.

Es ist sicher einzusehen, daß die Zufälligkeit nicht sehr hoch sein kann, es sei denn, die oben genannte und noch nicht erklärte Ausgangszahl ist zufällig.

Die erste Ausgangszahl hängt vom »Argument« des RND(X)-Befehls ab, das heißt vom Wert X, der in der Klammer dahinter steht. Es gibt drei Möglichkeiten für das Argument:
- eine positive Zahl (egal, welcher Wert)
- eine negative Zahl
- die Zahl 0

Eine positive Zahl
zum Beispiel RND (1) oder RND(56) nimmt als Samen die Zahl 0.811635157, die beim Einschalten des Computers als 5-Byte-Gleitkommazahl in die Speicherzellen 139 bis 143 geschrieben worden ist. In den fünf Zellen stehen die Zahlen 128, 79, 199, 82, 88.

Daraus folgt aber, daß nach dem Einschalten des Computers mit RND(1) immer dieselbe Sequenz von Zufallszahlen erzeugt wird. Schalten Sie bitte den Computer aus und ein und geben Sie ein:

```
10 PRINT RND(1):GOTO 10
```

Notieren Sie die ersten paar Zahlen und wiederholen Sie mit Aus-/Einschalten die Prozedur. Sie werden immer dieselben Zahlen sehen.

Zum Austesten von Programmen mit bekannten Zahlensequenzen ist diese Methode sicher wichtig, aber echte Zufallszahlen sind das nicht!

Eine negative Zahl
zum Beispiel RND(-1) oder RND(-95) bringt als erstes das Argument selbst (in meinem Beispiel -1 oder -95) als Gleitkommazahl in die Speicherzellen 139 bis 143, von wo sie als Samen den schon beschriebenen Manipulationen unterworfen wird. Nur – mit einem bestimmten negativen Argument erhalten Sie immer dieselbe Zufallszahl. Probieren Sie es aus:
PRINT RND(-2) ergibt immer dieselbe Zahl.

Es mag Fälle geben, wo die Vorgabe des allerersten Samens wünschenswert ist. Ich will aber von zufälligen Zahlen sprechen. Wir können diese Methode des negativen Arguments dadurch verbessern, daß wir als Argument selbst eine Zufallszahl nehmen.

Als derartige Zahl bietet sich der Wert der inneren Uhr TI an, die beim Einschalten des Computers losläuft und 60mal in der Sekunde weitergestellt wird. Da kein Mensch wissen kann, welchen Wert die Uhr TI gerade hat, kommt der Befehl RND(-TI) dem absoluten Zufall schon sehr nahe.

Das Argument (0)
verwendet eine andere Methode. Als Samen nimmt er eine sich ständig ändernde Zahl, die beim VC 20 aus vier Registerinhalten des VIC-Interface-Bausteins genommen werden. Beim C 64 wird es ähnlich gemacht, nur ist der VIC-Baustein ein anderer Typ.

Mit derselben Methode nach dem Einschalten wie im ersten Fall oben können Sie das leicht überprüfen.

Ich habe eingangs zitiert, daß RND(X) eine Zahl zwischen 0 und 1 erzeugt; das gilt aber nur für ein positives Argument. Wenn Sie hingegen eine Zufallszahl innerhalb eines ganz bestimmten Bereiches brauchen, müssen Sie anders vorgehen.

Kochrezept Nr. 1

Mit folgender Formel ist der Zahlenbereich beliebig vorgebbar:

```
X=(RND(1)*A)+B
```

Die Zahl A gibt einen Bereich von 0 bis A vor.

Die Zahl B legt den untersten Wert des Bereiches fest.

Beispiele:

```
10 PRINT (RND(1)*6)+1:GOTO 10
```
erzeugt Zahlen von 1 bis 6

```
10 PRINT (RND(1)*52)+1:GOTO 10
```
erzeugt Zahlen von 1 bis 52

```
10PRINT (RND(1)*6)+10:GOTO 10
```
erzeugt Zahlen von 10 bis 16

Mit dem Vorschalten der Funktion INT vor den Befehl RND werden die Zufallszahlen auf ganze Zahlen beschränkt.

```
10 PRINT INT (RND1)*6)+10:GOTO 10
```

Noch einmal: Zufallszahlen innerhalb bestimmter Zahlenbereiche sind gekoppelt mit einem positiven Argument. Wir haben aber gesehen, daß gerade so keine echten Zufallszahlen erzeugt werden. Deshalb brauchen wir noch ein zweites Kochrezept.

Kochrezept Nr. 2

Wenn Sie in einem Programm nach dem Einschalten des Computers immer neue Zufallszahlen brauchen, ist es empfehlenswert, für die allererste Zufallszahl RND(-TI) oder RND(0) zu verwenden, dann aber mit RND(1) fortzufahren.

Dasselbe gilt, wenn ein Programm wegen INPUT oder WAIT eine Pause hat. Nach der Pause sollte zuerst ein neuer Ausgangswert genommen werden.

Als letztes will ich noch beschreiben, wie man Zufallszahlen innerhalb von Maschinenprogrammen erzeugen kann.

Im Betriebssystem steht natürlich eine Routine für den Befehl RND. Im C 64 beginnt sie ab 57495 ($E097), im VC 20 ab 57492 ($E094).

Der Ausgangswert (Samen) wird dabei aus dem Gleitkomma-Akkumulator Nr. 1 geholt, dessen Vorzeichen oder Wert 0 das weitere Vorgehen der Routine bestimmt.

Sie müssen also den Samen in den Akkumulator Nr. 1 laden und dann mit JSR auf die RND-Routine springen. Als Resultat können Sie einen oder mehrere Werte der Zellen 140 bis 143 verwenden und nach Belieben weiterverarbeiten.

lich den Inhalt des Zeigers in 122 und 123 durch POKE verändern. Aber was dann? Jeder nachfolgende Basic-Befehl löst natürlich wieder die normale Übersetzungsroutine aus und unser schöner POKE ist für die Katz.

Wie ein Wedge in Maschinensprache gemacht wird, hat Christoph Sauer im VC 20-Kurs - 64'er, Ausgabe 9/84 - beschrieben. Allerdings ist das Beispiel für Anfänger nicht verständlich, was mich zu der Überzeugung bringt, daß die CHARGET-Routine und ihre Anwendung einen eigenen Aufsatz wert wäre.

Adresse 139 bis 143 ($8B bis $8F)

Wert der RND-Funktion als Gleitkommazahl

Mit dem Befehl RND(X) kann bekanntlich eine Zufallszahl erzeugt werden. Was das bedeutet und wie »zufällig« diese Zahlen sind, können Sie dem Texteinschub Nr. 13 »Wie zufällig sind Zufallszahlen?« entnehmen.

Beim Einschalten des Computers werden die Zahlen 128, 79, 199, 82 und 88 in diese Speicherzellen geschrieben. Mit der folgenden Zeile können Sie das gleich nach dem Einschalten des Computers leicht überprüfen.

```
FOR X=139 TO 143:PRINT
PEEK(X):NEXT
```

Nach den Manipulationen des RND-Befehls wird das Resultat wieder in die Zellen 139 bis 143 als neuer Ausgangswert (seed) für den nächsten RND-Befehl gebracht.

Diese fünf Zahlen stellen eine Gleitkommazahl dar. Ihre Form entspricht dabei der Aufteilung, wie sie auch im Gleitkomma-Akkumulator (97 bis 101) verwendet wird.

Eine Abfrage dieser Zahlen aus den Zellen 139 bis 143 ist natürlich möglich, aber nicht ergiebig, weil das Resultat von RND(X) direkt als Zahl verfügbar ist, während die 5 Byte erst in eine brauchbare Zahl umgerechnet werden müßten. Eine Änderung durch POKEn neuer Werte in diese Speicherzellen geht leider nicht.

Adresse 144 ($90)

Statusvariable ST

Diese Adresse enthält ein Byte, welches mit der Statusvariablen ST von Basic identisch ist. Diese reservierte Variable ist im Texteinschub Nr. 14 »ST-atus« näher beschrieben.

Alle Routinen des Betriebssystems, die mit Ein- und Ausgabe zu tun haben, benutzen diese Speicherzelle zum Abspeichern und Abfragen des Status der Ein-/Ausgabeoperationen.

Genauer gesagt, alle Ein-/Ausgabeoperationen, die mit der Datasette und mit dem Floppy-Gerät beziehungsweise dem Drucker zu tun haben, benutzen die Adresse 144. Im Fachjargon sprechen wir vom Kassetten-Port und vom seriellen Port.

Der dritte Anschluß des Computers, nämlich der RS232 oder User-Port, benutzt für den Status die Speicherzelle 663.

Kassette:		
Bit 2	(Wert 4)	Kurzer Block
Bit 3	(Wert 8)	Langer Block
Bit 4	(Wert 16)	Lesefehler (nicht korrigierbar)
Bit 5	(Wert 32)	Prüfsummenfehler
Bit 6	(Wert 64)	File-Ende
Bit 7	(Wert 128)	Band-Ende
Floppy/Drucker:		
Bit 0	(Wert 1)	Fehler beim Schreiben
Bit 1	(Wert 2)	Fehler beim Lesen
Bit 6	(Wert 64)	Daten-Ende
Bit 7	(Wert 128)	»Device Not Present«-Fehler

Jedes Bit der Zelle 144 hat eine eigene Bedeutung wie folgt.

Alle nicht aufgeführten Bits sind nicht benutzt.

Diese Speicherzelle beziehungsweise die Statusvariable ST kann recht nützlich sein. Einige Kochrezepte dafür werden im Texteinschub Nr. 14 behandelt.

Adresse 145 ($91)

Zwischenspeicher für Abfrage der STOP-Taste

In den Bildern 13 und 14 ist dargestellt, wie die Tasten des VC 20 und des C 64 miteinander über eine Matrix verbunden sind.

60mal in der Sekunde unterbricht der Computer seine Arbeit, merkt sich, wo er gerade ist und fragt dann unter anderem, ob die STOP-Taste gedrückt worden ist. Dadurch wird erreicht, daß die STOP-Taste jederzeit Priorität hat.

Die Abfrage geht so vonstatten, daß das Betriebssystem über das im Bild 13 und 14 gezeigte Spaltenregister 56320 (beim VC 20: 37152) diejenige Tastenspalte anwählt, in welcher sich die STOP-Taste befindet. Aus Bild 13 und 14 sehen wir, daß dies die Spalte mit der Codenummer 127 beziehungsweise 247 ist. Ist in dieser Spalte eine Taste gedrückt, wird an ihrer Stelle eine Null in das Auslese-Register 56321 (VC 20: 37153) geschrieben. Die dadurch entstandene Dualzahl wird in die Speicherzelle 145 gebracht.

Es ist sicher verständlich, daß auf diese Weise nicht nur die STOP-Taste, sondern alle Tasten der Spalte 127 (247) abgefragt werden können. Ein kleines Demonstrationsprogramm kann das beweisen:

```
10 PRINT PEEK (656321);
PEEK (145)
20 GOTO 10
```

Beim VC 20 ist statt 56321 natürlich 37153 einzusetzen.

Das Zahlenband kann durch die Tasten der genannten Spalte - und nur durch diese - beeinflußt werden.

Adresse 146 ($92)

Zeitkonstante beim Lesen vom Band

Die Speicherzelle enthält eine vom Betriebssystem einstellbare Zahl, welche die kleinen Unterschiede in der Aufnahmegeschwindigkeit ausgleicht, die bei verschiedenen Datasetten vorkommen können.

Diese Zeitkonstante steht im Zusammenhang mit der Zahl, die in den Speicherzellen 176 und 177 steht.

Eine Veränderung der Konstante in Basic ist nicht möglich.

Adresse 147 ($93)

Flagge für LOAD oder VERIFY

Diese Flagge dient dem Betriebssystem, um zu unterscheiden, ob eine LOAD-Operation nur LOADen oder aber VERIFYen soll.

Sie ist identisch mit der Flagge des Basic-Übersetzers in Speicherzelle 10. Genauere Hinweise bitte ich der Beschreibung von Zelle 10 zu entnehmen.

Adresse 148 ($94)

Flagge für Floppy/Drucker-Ausgang

Das Betriebssystem benutzt diese Speicherzelle, um anzuzeigen, daß ein Zeichen im Ausgabepuffer steht, welches zum Floppy-Gerät oder zum Drucker geleitet werden soll. Diese Flagge setzt alle am seriellen Port angeschlossenen Geräte in den Zustand »Listen«, das heißt bereit zu sein, Daten aufzunehmen.

Adresse 149 ($95)

Zeichen im Ausgabepuffer

In dieser Speicherzelle wird das Zeichen abgelegt, welches als nächstes über den Serial-Port zum Floppy-Gerät oder zum Drucker transportiert wird, sobald die Flagge in 148 die Bereitschaft anzeigt.

Adresse 150 ($96)

Arbeitsspeicher für die Band-Leseroutine

Diese Speicherzelle wird zur Zwischenspeicherung von Daten beim Lesen einer Kassette benutzt.

Adresse 151 ($97)

Zwischenspeicher des X-Registers

Maschinen-Programmierer kennen das X-Register des Mikroprozessors. Beim Lesen eines Zeichens von der Datasette wird der Inhalt des X-Registers in dieser Adresse zwischengespeichert.

Adresse 152 ($98)

Anzahl der offenen Files

Ein File, oder auf Deutsch gesagt, eine Datei, wird mit dem Befehl OPEN eröffnet. Nach OPEN folgt die Nummer der Datei; sie ist beliebig wählbar bis maximal 255. Als zweites folgt die Nummer des Gerätes, mit dem die Verbindung hergestellt werden soll.

Es ist erlaubt, mehrere Dateien gleichzeitig geöffnet zu halten, vorausgesetzt die Nummern der Dateien sind verschieden.

In Speicherzelle 152 wird festgehalten, wieviel Dateien gleichzeitig geöffnet sind. Dieses kleine Programm zeigt es uns deutlich:

```
10 FOR K=10 TO 22
20 PRINT PEEK (152),K
30 OPEN K,0
40 NEXT K
```

Mit der FOR...NEXT-Schleife der Zeilen 10 und 40 eröffnen

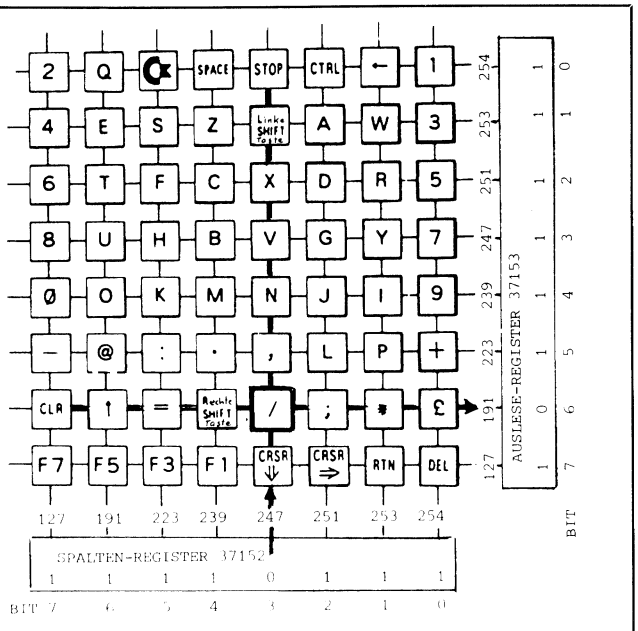


Bild 13. Tastenanordnung des VC 20

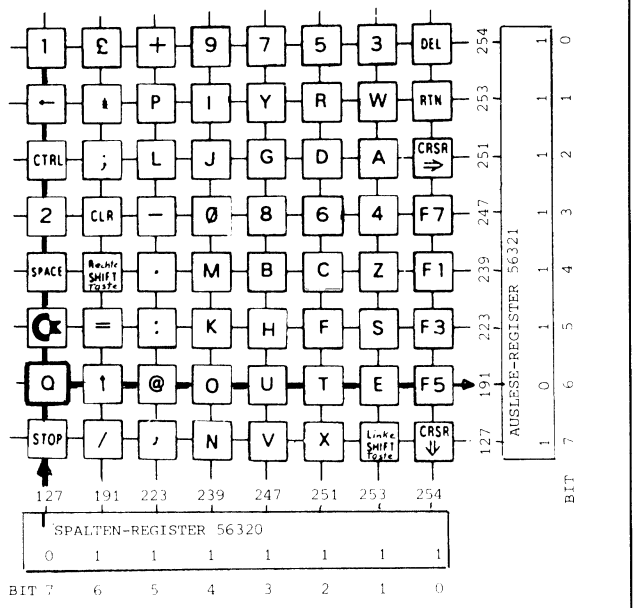


Bild 14. Tastenanordnung des C 64

wir 13 Dateien hintereinander, und zwar - wie Zeile 30 uns deutlich macht - mit der Tastatur. Die Tastatur hat die Nummer 0, der Drucker die Nummer 4, das Floppy-Gerät die Nummer 8 und die Datasette die Nummer 1. Ich habe die Tastatur gewählt, obwohl es keinen Sinn ergibt, weil sie die vielen Eröffnungen ohne zu unterbrechen akzeptiert.

Nach RUN sehen wir links untereinander den Inhalt von 152, also die Anzahl der eröffneten Dateien. Rechts steht jeweils die Nummer der eröffneten Datei.

Nach der 10. Datei bricht das Programm ab und druckt uns die Fehlermeldung TOO MANY FILES aus.

Das heißt es sind gleichzeitig nur 10 Dateien betreibbar. Wenn wir oben in Zeile 10 die Zahl 22 durch 19 ersetzen, läuft das Programm fehlerfrei.

Eine Datei, die unter einer bestimmten Nummer eröffnet ist, kann nicht noch einmal eröffnet werden. Fügen Sie bitte dem Programm noch die folgende Rücksprungzeile hinzu:

```
50 GOTO 10
```

In der 10. Zeile sehen wir jetzt die 10 als Inhalt von 152 und als neue Dateinummer (Schleifenvariable K) wieder die 10. Das Programm bleibt aber stehen und meldet FILE OPEN. Es hat recht, denn die Datei 10 ist bereits als erste eröffnet, aber nicht wieder geschlossen worden.

Das Betriebssystem macht das so, daß jede der Dateinummern in eine Tabelle geschrieben wird, die in den Speicherzellen 601 bis 610 stehen. Bei jedem OPEN-Befehl wird dort nachgeschaut, ob die Filenummer existiert. Wenn ja, wird die Fehlermeldung FILE OPEN ERROR ausgegeben. Bei jedem CLOSE-Befehl wird die entsprechende Nummer aus der Tabelle gelöscht.

Wir können aber auch eine 0 in die Speicherzelle 152 POKEn, wodurch dem Betriebssystem vorgegaukelt wird, daß keine Datei eröffnet ist. Schieben Sie im Programm einfach die Zeile ein:

```
45 POKE 152,0
```

und das Programm läuft ewig weiter.

Die Speicherzelle 152 ist also der Wächter über die Anzahl der eröffneten Dateien. Steht sie auf 0, dann wird eine Neueröffnung am Anfang der Tabelle ab 601 eingetragen. Die Tabelle ihrerseits ist der Wächter über Exklusivität der Dateinummern. Ich zeige Ihnen das noch genauer, wenn wir zu 601 kommen.

Sie werden vielleicht fragen, warum ich das so ausführlich beschreibe. Nun, in einem Programm kann es sicher sehr nützlich sein, die Zelle 152 mit PEEK nach der Datei-Lage abzufragen und entsprechend Maßnahmen zu treffen, ehe die Fehlermeldung das Programm abbricht.

Mit POKE 152,0 aber müssen Sie aufpassen. Es ersetzt nämlich nicht (!!) den CLOSE-Befehl. Probieren Sie es aus: Um das kleine Programm oben per Drucker auszudrucken, brauchen wir:

```
OPEN 1,4: CMD 1: LIST
```

Wenn Sie jetzt die Zeile 152 auf 0 POKEn und dann LIST eintippen, wird trotzdem wieder auf dem Drucker gelistet und nicht auf dem Bildschirm. Die vorgeschriebene Schließmethode mit

```
PRINT #1:CLOSE 1
```

geht jetzt aber auch nicht mehr, denn das Betriebssystem ist ja im Glauben, daß keine Datei eröffnet ist - schöner Schlamassel!

Erst eine Neueröffnung bringt alles wieder in die Reihe. Also Vorsicht mit der Anwendung der Speicherzelle 152. Eine Möglichkeit, alle Dateien auf einen Schlag zu schließen, gibt es aber doch.

SYS 65511 besorgt das sowohl beim C 64 als auch beim VC 20.

Adresse 153 ($99)

Nummer des Eingabe-Gerätes

Das Betriebssystem verwendet diese Speicherzelle, um festzuhalten, welches Gerät zur Eingabe verwendet werden soll.

Die Nummern sind wie folgt festgelegt:

```
0     = Tastatur
1     = Datasette
2     = RS232 (User-)Port
3     = Bildschirm
4, 5  = Drucker
8-11  = Floppy-Laufwerke
```

Nach dem Einschalten oder nach RESET des Computers steht in 153 eine 0 (Tastatur). Nach jedem Einsatz eines anderen Gerätes wird diese Speicherzelle wieder auf 0 gesetzt, so daß wir immer die Tastatur zur Verfügung haben.

Für Maschinenprogrammierer ist diese Adresse sicherlich wertvoll. Die Routine, welche die Eingabegeräte festlegt, sobald der Befehl INPUT# beziehungsweise GET# ausgeführt wird, heißt CHKIN und beginnt beim C 64 ab Adresse 61966 ($F20E), beim VC 20 ab 62151 ($F2C7).

Für Basic-Programmierer habe ich in der Literatur nur eine Anwendung gefunden, und die wurde bereits bei der Besprechung der Speicherzelle 19 angekündigt.

Es ist dies eine MERGE-Routine. Leider funktioniert dieses Verfahren nicht bei dem 1541-Floppy-Laufwerk. Erfunden wurde die Routine von Brad Templeton und ist von Jim Butterfield unter dem Namen »Magic Merge« für den VC 20/C 64 adaptiert worden. Ich gebe zu, in der Zwischenzeit sind noch andere, vielleicht auch kürzere MERGE-Routinen veröffentlicht worden. Aber diese hier verwendet gleich drei interessante Ingredienzen, nämlich die Speicherzellen 19 und 153 und außerdem die sogenannte »Dynamische Tastenabfrage«. Wer die letztere nicht kennt, sollte sich zum Verständnis den Texteinschub Nr. 15 gleichen Namens ansehen.

Ein MERGE (deutsch: zusammenführen, verschmelzen) be-

steht darin, ein auf Band gespeichertes Programm zu einem im Computer stehenden anderen Programm so dazuzuladen, daß dieses nicht überschrieben, sondern ergänzt wird. Wichtig ist dabei, daß das Programm vom Band höhere Zeilennummern hat als das Programm im Computer. Außerdem muß das Programm auf dem Band als Datei gespeichert sein. Das wird so erreicht:

1. Programm eintippen
2. Direkt eingeben:
   OPEN 1,1,1,"Name":
   CMD1:LIST
3. Erst wenn READY kommt, direkt eingeben
   PRINT#1:CLOSE1

Damit ist das Programm auf dem Band gespeichert. Nun kommt der eigentliche MERGE-Vorgang.

4. Es steht ein Programm im Computer
5. Band mit dem Programm »Name« einlegen
6. Direkt eingeben:
   POKE 19,1:OPEN 1
7. Sobald READY erscheint, Bildschirm löschen (SHIFT-CLR).
8. Dreimal Cursor-Down
9. Direkt eingeben:
   PRINT CHR$(19):
   POKE 198,1:
   POKE 631,13:
   POKE 153,1
10. Das Band beendet den Ladevorgang mit einer Fehlermeldung, die wir ignorieren.
11. Nach CLOSE 1 sind beide Programme zusammengefügt.

Wie gesagt, Schritt 6 verwendet Zelle 19 (bitte dort nachlesen), Schritte 8 und 9 sind die »Dynamische Tastenabfrage«, und Schritt 9 verwendet zusätzlich die hier zur Diskussion stehende Speicherzelle 153, um die Datasette als Eingabegerät zu definieren.

Adresse 154 ($9A)

Nummer des Ausgabe-Gerätes

Diese Speicherzelle entspricht der Zelle 153, nur steht hier die Nummer des Gerätes, über das die Ausgabe läuft.

Nach dem Einschalten und nach Ausgabeoperationen wird der Wert immer auf 3 gesetzt. Das ist entsprechend der oben genannten Zuordnung der Bildschirm.

Für Maschinenprogrammierer sei erwähnt, daß Basic bei den Befehlen PRINT# oder CMD die Routine CHKOUT einsetzt, welche die Adresse 154 belegt. Sie steht im C 64 ab Adresse 62032 ($F250), im VC 20 ab 62217 ($F309).

Adresse 155 ($9B)

Fehlerkontrolle bei Bandoperationen

Die Commodore-Datasette ist deswegen so zuverlässig, weil sie mehrere Methoden zur Fehlererkennung beziehungsweise Korrektur von Lese- und Schreibfehlern verwendet.

Eine der Methoden ist die sogenannte Parity-Prüfung. Sie ist nichts anderes als eine Quersummenbildung der einzelnen Stellen jedes Bytes, deren Resultat überprüft wird.

Die Speicherzelle 155 wird bei dieser Parity-Prüfung eingesetzt.

Adresse 156 ($9C)

Flagge für korrektes Byte vom Band

In dieser Speicherzelle wird zwischengespeichert, ob das vom Band gelesene Byte die Prüfungen bestanden hat, also richtig ist oder nicht.

Adresse 157 ($9D)

Flagge für Meldungen

Man muß zwischen zwei Arten von Meldungen unterscheiden:

Meldungen des Betriebssystems
Meldungen des Basic-Übersetzers

Die Meldungen des Betriebssystems kennen wir als Angaben zum Ablauf, wie SEARCHING FOR, FOUND, PRESS PLAY ON TAPE und so weiter. Normalerweise nicht bekannt ist die Meldung I/O ERROR #, wobei nach dem Zeichen # Zahlen von 0 bis 29 stehen können. Diese Zahlen beziehen sich auf Meldungen des Übersetzers (Interpreter), die ausschließlich Fehlermeldungen sind. Das mag verwirrend klingen, klärt sich aber sofort. Die Flagge in 157 kann vier Werte annehmen: 0, 64, 128 und 192.

1. Der Wert 0 unterdrückt alle Meldungen des Betriebssystems. Dieser Modus tritt nach RUN beim Ablauf eines Programms ein.
2. Der Wert 64 läßt nur Fehlermeldungen des Betriebssystems zu. Dieser Modus ist normalerweise nicht vorgesehen, kann aber künstlich erzeugt werden.
3. Der Wert 128 unterdrückt die Fehlermeldung des Betriebssystems. Dieser Modus entspricht dem Normalfall.
4. Der Wert 192 läßt alle Meldungen zu. Auch dieser Modus ist nur künstlich herzustellen.

Das folgende Beispiel macht das deutlich. Geben Sie direkt ein:

```
POKE 157,0:LOAD "$",9
```

Wir versuchen, vom Gerät mit der Nummer 9, das ist eine zweite Floppy, die Directory zu laden. Wir erhalten entsprechend Punkt 1 nur die Meldung des Übersetzers »? DEVICE NOT PRESENT«.

Verändern wir den POKE-Befehl für Punkt 2:

```
POKE 157,64:LOAD"$",9
```

Wir erhalten jetzt »I/O ERROR #5 ? DEVICE NOT PRESENT«.

```
POKE 157,128:LOAD"$",9
```

ergibt die Meldung »SEARCHING FOR $
? DEVICE NOT PRESENT«.

Schließlich nehmen wir noch den letzten Fall:

```
POKE 157,192:LOAD"$",9
```

Jetzt erhalten wir alles:
»SEARCHING FOR $
I/O ERROR #5
? DEVICE NOT PRESENT«

Da die Fehlermeldung des Betriebssystems und die zugehörigen Nummern in keinem Handbuch erwähnt sind, habe ich sie interessehalber in der folgenden Tabelle zusammengefaßt.

#	MELDUNG (ERROR)
1	TOO MANY FILES
2	FILE OPEN
3	FILE NOT OPEN
4	FILE NOT FOUND
5	DEVICE NOT PRESENT
6	NOT INPUT FILE
7	NOT OUTPUT FILE
8	MISSING FILE NAME
9	ILLEGAL DEVICE NUMBER
10	NEXT WITHOUT FOR
11	SYNTAX
12	RETURN WITHOUT GOSUB
13	OUT OF DATA
14	ILLEGAL QUANTITY
15	OVERFLOW
16	OUT OF MEMORY
17	UNDEF'D STATEMENT
18	BAD SUBSCRIPT
19	REDIM'D ARRAY
20	DIVISION BY ZERO
21	ILLEGAL DIRECT
22	TYPE MISMATCH
23	STRING TOO LONG
24	FILE DATA
25	FORMULA TOO COMPLEX
26	CAN'T CONTINUE
27	UNDEF'D FUNCTION
28	VERIFY
29	LOAD

Texteinschub Nr. 14
ST-atus

Neben den Befehlen (wie PRINT) und den Funktionen (wie COS) hat Basic auch noch drei fest definierte Variable, nämlich TI, TI$ und ST.

Von den dreien ist ST wohl am seltensten anzutreffen, Grund genug, hier ein wenig darüber zu berichten.

Der Anlaß ist natürlich, daß der Wert von ST immer in der Speicherzelle 144 steht, die ja hier vorkommt.

Bei der Beschreibung wurde schon erwähnt, daß ST den Status nach der letzten Ein- beziehungsweise Ausgabeoperation angibt, beschränkt allerdings nur auf Operationen mit der Datasette und der an einem gemeinsamen Ausgang angeschlossenen Floppy und Drucker.

Dementsprechend zeigt die Tabelle bei der Speicherzelle 144 diese beiden Fälle.

Wichtig ist noch zu erwähnen, daß nicht nur die in der Tabelle gezeigten Zahlen für ST auftreten, sondern auch Kombinationen davon. So ergibt zum Beispiel ein zu kurzer Block (4) und ein gleichzeitig aufgetretener Prüfsummenfehler (32) einen Wert von 36.

Kassettenoperationen

Zuerst testen wir mit einem Datei-Programm auf »File-Ende«. Geben Sie bitte folgendes Programm ein:

```
10  OPEN 1,1,1,"DATEI"
20  PRINT#1, "QWERTY"
30  CLOSE 1
40  END
```

Zur Erinnerung: Nach dem OPEN-Befehl folgt zuerst die Nummer der Datei (ich nehme hier 1), dann die Gerätenummer (1= Datasette) und schließlich die Sekundäradresse (1= Schreiben).

Jetzt kommt der Lesevorgang:

```
 50 OPEN 2,1,0"DATEI"
 60 FOR K=1 TO 10
 70 GET#2,A$
 80 PRINT A$;ST
 90 NEXT K
100 CLOSE 2
```

In Zeile 50 eröffnen wir wieder eine Datei (diesmal Nummer 2) für die Datasette, jetzt aber zum Lesen (Sekundäradresse=0). Die Schleifen der Zeilen 60 und 90 schreiben uns 10mal ein Zeichen (A$) und den Wert von ST auf den Bildschirm.

Jetzt geht es los. Mit RUN starten wir den ersten Teil des Programms. Nach dem Schreibvorgang und der READY-Meldung (nach Zeile 40) müssen Sie das Band zurückspulen und mit GOTO 50 ab Zeile 50 weiterfahren. Jetzt wird die Datei gelesen.

Wir erhalten untereinander die sechs Buchstaben von Zeile 20, daneben für ST lauter Nullen. Am Ende allerdings erscheint eine 64. Das ist der in der Tabelle angegebene Wert von ST für »File-Ende«.

Da die FOR-NEXT-Schleife zu lang ist, schießen wir mit dem Lesen über das File-Ende hinaus. Normalerweise kennen wir natürlich die Länge einer Datei nicht. Deshalb ist es besser, mit GOTO zurückzuspringen und das File-Ende abzufragen.

Löschen Sie bitte Zeile 60 und 90 und fügen Sie als Rücksprung und Prüfung ein:

```
85  IF ST=64 THEN 100
90  GOTO 70
```

Statt nach ST können wir natürlich genausogut nach PEEK(144) fragen.

Ein erneutes GOTO 50 bringt das erwünschte Resultat.

Um den vorhin schon erwähnten »kurzen Block« zu sehen, müssen wir einen entsprechenden Fehler künstlich erzeugen.

Löschen Sie bitte den ersten Teil des Programms bis einschließlich Zeile 40. Wir behalten also nur den Leseteil ab Zeile 50. Dann laden wir dieses Programm (Band vorher am besten wieder zurückspulen) mit SAVE "DATEI" nicht als Datei, sondern als ganz gewöhnliches Programm. Wenn es geladen ist, bitte das Band wieder zurückspulen.

Mit RUN starten wir jetzt das Lese-Programm, welches eine Datei sucht, aber nur ein Programm findet, allerdings mit dem richtigen Namen. Natürlich findet es einen Fehler und wir erhalten als Ausdruck:

```
 36   oder manchmal auch 4
 64                      64
```

Die Zahl 36 entsteht aus 32+4, das bedeutet Prüfsummenfehler + Block zu kurz. Danach folgt wie vorher das File-Ende.

Die normale Blocklänge entspricht der Länge des Bandpuffers, in den die Datasette einspeichert. Er ist 191 Byte lang. In unserem Fall war offenbar der Block nicht ganz voll.

Der Prüfsummenfehler tritt dann auf, wenn eine der Überprüfungen von Kassettenoperationen einen Fehler gefunden hat. Der Blockfehler, auch der des zu langen Blocks, interessiert wohl weniger. Aber ein durch die Prüfungen gefundener Fehler könnte frühzeitig, noch vor dem Ausstieg des Programms, entdeckt, abgefangen und ausgenutzt werden.

Die Formel dafür, ins obige Programm eingebaut, ist:

```
85 IF ST<64 OR ST>8 THEN..(zum Beispiel LIST)
```

Statt LIST kann man natürlich auch etwas anderes nehmen.

Floppy-Operationen

Bei der Floppy bedeutet ST=64, »Daten-Ende«, das ist etwa dasselbe wie bei der Datasette. Um es zu überprüfen, nehmen wir dasselbe Programm wie vorher, nur müssen wir die Datei-Zeilen der Floppy anpassen. Das sieht dann so aus:

```
 10 OPEN1,8,1"DATEI,S,W"
 20 PRINT#1,"QWERTY"
 30 CLOSE 1
 40 END
 50 OPEN 2,8,0"DATEI,S,R"
 60 FOR K=1 TO 10
 70 GET#2,A$
 80 PRINT A$,ST
 90 NEXT K
100 CLOSE 2
```

Das Ergebnis sieht hier so aus:

```
64
66
66
```

Die 64 ist natürlich wie erwartet der »Wert« für Daten-Ende. Die 66 ist 64 + 2, entstanden dadurch, daß wir über das Daten-Ende hinausgelesen haben. Die 2 bedeutet »Fehler beim Lesen« (in den englischen Beschreibungen heißt es »Read Time Out«). Ähnliches gilt für ST=1, das bedeutet »Fehler beim Schreiben« (englisch: Write Time Out), nur weiß ich leider nicht, wie es vorzuführen ist. Wie bei der Datasette kann das Überlesen natürlich mit der Abfrage IF ST=64 THEN...und GOTO...gestoppt werden.

Interessant ist noch der Status beim Fehler »DEVICE NOT PRESENT«, den wir dadurch erzeugen, daß wir ein Programm oder die Directory von der Diskette laden wollen, ohne daß dieses Gerät angeschlossen oder eingeschaltet ist. Nach der Fehlermeldung geben wir direkt ein:

```
PRINT ST oder PRINT PEEK(144)
```

und wir erhalten die Zahl 128.

Wie man allerdings in einem Basic-Programm durch Abfrage von ST=128 die Fehlermeldung »Device Not Present« und den dann folgenden Programmabbruch vermeiden kann, bedarf einer gesonderten Maßnahme:

Es gibt zwei Speicherzellen - 768 und 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. Das geänderte Programm sieht dann so aus:

```
 10 POKE 768,61                  Fehlermeldung abschalten
 20 OPEN 1,8,15                  Gerät ansprechen
 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 des DEMO
100 PRINT »GEREAT EINSCHALTEN«
110 GET A$ : IF A$ = "" THEN 10 Warteschleifen

120 GOTO 10                      neuer Versuch
```

Drucker-Operationen

Für den Drucker sieht die Änderung fast genauso aus. Die einzige Änderung ist in den Zeilen 20 und 30

```
 20 OPEN 1,4
 30 PRINT#1
```

Unter bestimmten Umständen kann das Verbiegen des Zeigers in 768 entfallen.

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.

Texteinschub Nr. 15
Dynamische Tastenabfrage

Jedesmal, wenn Sie eine Taste drücken, wird der ASCII-Codewert des Zeichens oder der Funktion dieser Taste ermittelt und im »Tastaturpuffer« gespeichert. Dieser Pufferspeicher liegt in den Speicherzellen 631 bis 640.

Die eigentliche Abfrage und Umcodierung eines Tastendrucks habe ich im Kurs »Alle Tasten-, Zeichen- und Steuercodes« in den Ausgaben 4/84 bis 7/84 beschrieben. Er ist auch im 64er-Sonderheft Nr. 2/86 enthalten. Die ASCII-Codetabelle finden Sie da auch. Sie ist natürlich in allen Handbüchern enthalten, leider aber nicht immer ganz vollständig.

Zuerst will ich Ihnen die Arbeitsweise des Tastaturpuffers zeigen.

Der Computer holt sich den ASCII-Codewert aus dem Tastaturpuffer und wenn gerade kein Programm läuft, druckt er das Zeichen auf den Bildschirm oder führt die Funktion der Taste aus. Das ist der oft zitierte »Direkt-Modus«.

Wenn aber ein Programm läuft, dann bleiben die Codezahlen im Puffer so lange stehen, bis der Computer fertig ist. Dann erst werden sie herausgeholt und verarbeitet. Das will ich Ihnen beweisen.

Tippen Sie im Direkt-Modus ein:

```
FOR K=1 TO 15000:NEXT K (RETURN)
```

Während diese an sich sinnlose Zeitschleife 15000mal im Kreis rumrennt, haben Sie genügend Zeit, mehrere Tasten zu drücken, zum Beispiel die erste Buchstabenreihe (QWERTYUIOP@*↑). Natürlich sehen Sie am Bildschirm gar nichts, denn das Programm der Schleife läuft ja noch. Sobald aber die Schleife zu Ende ist, erscheinen 10 der getippten Buchstaben. Quod erat demonstrandum!

Warum nur 10 Buchstaben? Nun, der Tastaturpuffer hat nur 10 Plätze, logisch?

Jetzt müssen wir noch eine weitere Speicherzelle ins Spiel bringen. In die Zelle 198 kann man nämlich eine Zahl hineinPOKEn, welche die Anzahl der Zeichen im Tastaturpuffer begrenzt.

Wiederholen Sie bitte das Experiment von oben, nur soll die direkt eingegebene Zeile erweitert werden:

```
FOR K=1 TO 15000:NEXT K: POKE 198,6 (RETURN)
```

Und siehe da, jetzt werden nur die sechs Buchstaben Q bis Y ausgedruckt. Diese Anwendung des Tastaturpuffers nutzen wir für das Kochrezept »Dynamische Tastenabfrage« aus. Allerdings müssen wir dazu prinzipiell die ASCII-Codewerte verwenden, so wie im nächsten Beispiel.

Löschen Sie bitte den Bildschirm und geben Sie ein (identisch für VC 20 und C 64):

```
10 PRINT CHR$(65)
20 PRINT CHR$(156)
30 PRINT CHR$(66)CHR$(13)CHR$(67)
```

65 ist der Code für A, 156 für die Farbe »purple«, 66 für B, 13 für »RETURN« und 67 für C. Das heißt also, daß wir zuerst ein A drucken, dann auf »purple« umschalten, darunter dann das B schreiben, einmal RETURN geben und dann noch das C folgen lassen.

Bild 15 zeigt den Ausdruck auf dem Bildschirm, wenn Sie diese Zeilen LISTen und dann mit RUN starten.

Zur Erklärung: Die Leerzeile zwischen A und B ist bedingt durch die PRINT-Anweisung in Zeile 20, welche nur die Farbe umschaltet. Obwohl die Codes für B und C zusammen in einer Zeile stehen, werden sie doch durch das »RETURN« CHR$(13) untereinandergesetzt.

Anstelle der 13 können Sie alle möglichen anderen Steuerfunktionen setzen. Bild 16 zeigt das Resultat von 17, nämlich »Cursor down«. Wenn Sie die 8 nehmen, können Sie den Zeichensatz nicht mehr ändern. Der Einsatz der gleichzeitig gedrückten SHIFT- und Commodore-Taste funktioniert erst nach CHR$(9) wieder.

Als nächstes wollen wir die ASCII-Codewerte und den Tastaturpuffer im »Programm-Modus« einsetzen.

Das Resultat von Bild 16 wollen wir jetzt durch POKEn der ASCII-Werte in den Tastaturpuffer wiederholen.

```
 5  POKE 198,5
10  POKE 631,65
20  POKE 632,156
30  POKE 633,66: POKE 634,17: POKE 635,67
99  END
```

Prinzipiell macht dieses Programm das gleiche wie das Programm in Bild 16. Trotzdem erhalten wir nach LIST und RUN ein anderes Ergebnis, nämlich das von Bild 17.

Ist das ein Fehler? Natürlich nicht. Nach RUN laufen zuerst einmal alle POKE-Befehle ab. Zeile 5 gibt an, wieviele Zeichen im Puffer stehen sollen. In Zeile 99 findet das Programm das ENDe und meldet sich mit READY. Jetzt erst wird im Tastaturpuffer nachgeschaut. Dort findet der Computer zuerst das A, dann »purpur«, dann das B, welches sofort neben das A gesetzt wird. Das ist auch logisch, denn es fehlt ja jede Angabe, eine Zeile tiefer zu gehen. Um das zu erreichen, müssen wir in der Zeile 10 den Codewert für RETURN einschieben:

```
10 POKE 631,65: POKE 632,13
```

Vorsicht! Sie müssen in Zeile 20 und 30 alle POKE-Adressen um 1 erhöhen und auch die Zahl in Zeile 5. Nehmen Sie 10, dann haben Sie Platz für Erweiterungen. So, jetzt LIST und RUN eingeben und es erscheint Bild 18 – und wir haben schon wieder ein Problem! Aber alles im Computer ist logisch! Nach dem A findet er den Wert für »RETURN«, also führt er den Befehl aus, auf dem der Cursor gerade steht. Der steht auf dem A. Da das kein gültiger Basic-Befehl ist, druckt der Computer die Fehler-Meldung und zeigt READY an. Danach allerdings macht er weiter wie oben.

Und jetzt kommt die ASCII-Zahl 141 (SHIFT-RETURN) voll zur Geltung. Diese Kombination nämlich setzt den Cursor an den Anfang der nächsten Zeile, ohne »RETURN« auszuführen.

Ersetzen Sie also die 13 in Zeile 10 durch 141, dann läuft's (Bild 19).

Es gibt übrigens noch eine interessante ASCII-Codezahl, die in keiner Tabelle steht, nämlich 131. Sie bedeutet dasselbe wie die geSHIFTete STOP-Taste, also die Funktion »LOAD+RUN«.

Wenn Sie diesen Code mit PRINT CHR$(131) ausprobieren, funktioniert er allerdings nicht. Deshalb steht er wohl auch nicht in den Tabellen.

In den Tastaturpuffer gePOKEt bringt er aber seine Wirkung.

Setzen Sie bitte in Zeile 30 an die Stelle von 67 die Zahl 131 und anstelle des C erscheint:

LOAD und PRESS PLAY ON TAPE. Gut, nicht wahr?

So, jetzt haben wir alle Zutaten für unser Kochrezept zusammen.

Löschen Sie bitte alles bisherige und tippen Sie ein:

```
10  PRINT CHR$(147)
20  FOR I=1 TO 5: PRINT I
30  FOR T=1 TO 500: NEXT T
40  NEXT I
```

Nach Löschen des Bildschirms (Zeile 10) drucken wir zum Ausschmücken die Zahlen 1 bis 5 untereinander (Zeile 20 und 40) und damit es nicht zu schnell geht, bremsen wir mit der Zeile 30.

```
50  PRINT "LIST" (Das ist natürlich sehr einfach, aber
    jetzt kommt's!)
60  POKE 198,5
70  POKE 631,145: POKE 632,145: POKE 633,145
80  POKE 634,13
90  END
```

Nach RUN erscheinen erst die fünf Zahlen, dann wird in einer neuen Zeile das Wort LIST geschrieben. Nach END wird erst (wie immer) eine Zeile ausgelassen, dann READY gedruckt und schließlich springt der Cursor an den Anfang der Zeile darunter. Während der Cursor anfangen will, drei Zeilen unter dem Wort LIST zu blinken, findet der Computer im Tastaturpuffer 3 den ASCII-Code für »Cursor up«. Also geht dieser auch drei Zeilen hoch und will jetzt auf dem Wort LIST blinken.

Damit wird es aber wieder nichts, denn im Puffer steht ja noch

der Code für RETURN (13). Das wird ausgeführt und zwar für das LIST. Es hat dieselbe Wirkung, als ob Sie direkt LIST tippen und danach auf die RETURN-Taste drücken, nämlich das Programm wird ausgeLISTet.

Alle sinnvollen Basic-Befehle, die Sie in der Zeile 50 PRINTen, werden durch diese dynamische Manipulation des Cursors ausgeführt.

Versuchen Sie es mit

```
50  PRINT →PRINT 16 * 35"
50  PRINT "LOAD"
50  PRINT "GOTO 10"
50  PRINT "RUN"
50  PRINT "RUN 50"
```

und falls Sie dieses kleine Programm geSAVEt haben

```
50  PRINT "SYS 64824"
50  PRINT "SYS 64763"
```

Die Kunst ist also, mit entsprechenden Codezahlen den Cursor an diejenige Stelle des Bildschirms zu bringen, wo innerhalb eines Programms eine geeignete Anweisung gedruckt worden ist. Man kann damit getrennte Programmteile nachladen, mit SYS-Befehlen Maschinenprogramme aufrufen, oder gar Programme durch sich selbst ändern lassen.

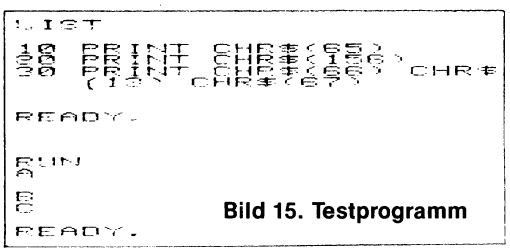


Bild 15. Testprogramm

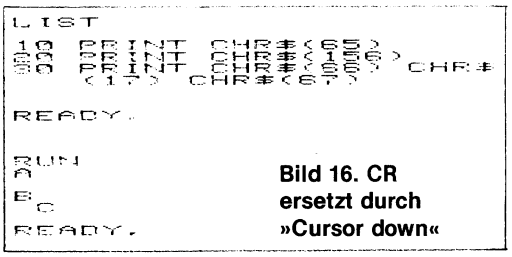


Bild 16. CR ersetzt durch »Cursor down«

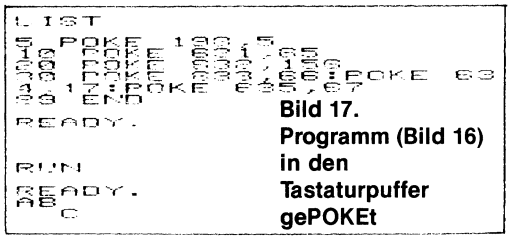


Bild 17. Programm (Bild 16) in den Tastaturpuffer gePOKEt

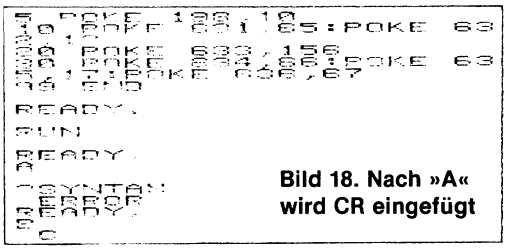


Bild 18. Nach »A« wird CR eingefügt

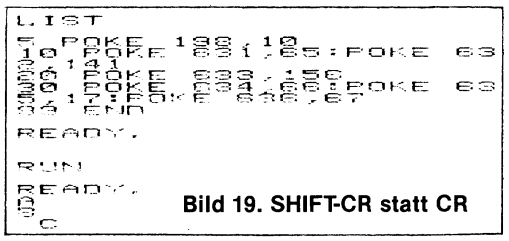


Bild 19. SHIFT-CR statt CR

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 = 1 092,26 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 zurückgesetzt. 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 Texteinschub Nr. 16 »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 und 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 1 019 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 Texteinschub 17 »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 noch andere Speicherzellen benötigt, die später an der Reihe sind, gehe ich auf die RS232-Schnittstelle erst bei der Behandlung der Speicherzelle 659 bis 673 näher ein.

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 gingen. Da derartige Prüfungen mit Parity-Bits (Quersummenprüfung) des öfteren erwähnt werden, gebe ich eine kurze Beschreibung des Prüfprinzips im Texteinschub 18 »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 und 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 und 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 ursprünglich verwendete Speicherbereiche zu bringen. Dazu sind aber noch einige andere Zellen notwendig, bis hin zu dem schon erwähnten Zeiger in 193 und 194.

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 und 173, nur zeigt er seinerseits auf die letzte Adresse des zu bewegenden Programms.

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).

Durch Verbiegen dieses Zeigers kann der Kassettenpuffer auf beliebige Plätze des Speichers, aber nicht unterhalb der Adresse 512 verschoben werden. Das kann durchaus sinnvoll sein, um im Kassettenpuffer abgelegte Maschinenprogramme vor Überschreiben durch Kassettenoperationen zu schützen.

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 18) 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.

Texteinschub Nr. 16
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 Spei-

cherzellen 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 Zählerprinzip (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 Countdown-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 Countdown 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 Nr. 17
Experimente mit dem Kassetten-Puffer

Die Speicherzellen von 828 bis 1 019 werden als »Kassetten-Puffer« bezeichnet.

Beim Speichern 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 #1,1
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 so lange 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 den 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 Byte, 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 noch mal 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#1,1
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 Speichern 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 Nr. 18
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.

Im binären Zahlensystem sieht das so aus:

DEZ	BINÄR	PARITYBIT	QUERSUMME
0	0000	0	0
1	0001	1	0
2	0010	1	0
3	0011	0	0
4	0100	1	0
.	.	.	.
.	.	.	.

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.

Texteinschub Nr. 19
Files - Geräte - Namen - Nummern

In den Handbüchern von Commodore und auch in anderen Beschreibungen wird von den Ein- und Ausgabe-Befehlen, wie zum Beispiel LOAD, SAVE, OPEN etc., leider ein recht verwirrendes Bild geboten. Ich beziehe mich dabei auf die hinter diesen Befehlen stehenden Ziffern und Namen.

In der Bezeichnung »File-Name« und »Geräte-Nummer« sind sich die Autoren noch weitgehend einig. Daß die dritte Angabe hinter den Befehlen aber Sekundär-Adresse, Command, Speicheradressen-Flag oder gar EOT-Flag genannt wird, muß den Computeramateur zwangsläufig verwirren.

Da wir im nebenstehenden Kurs gerade die Speicherzellen 183 bis 188 besprechen, die alle mit diesen Anhängseln der eingangs genannten Befehle zusammenhängen, ist dies eine gute Gelegenheit, etwas Systematik in die Angelegenheit zu bringen.

Als erstes gebe ich die Bezeichnungen der Befehle aus einem Commodore-Buch wieder, bevor die einzelnen Angaben im Detail diskutiert werden.

- LOAD "File-Name", Geräte-Nr., Speicheradressen-Flag
- SAVE "File-Name", Geräte-Nr., EOT-Flag
- VERIFY "File-Name", Geräte-Nr., Speicheradressen-Flag
- OPEN File-Nr., Geräte-Nr., Sekundär-Adresse, "File-Name, Typ, Modus"
- INPUT# File-Nr., Variable
- GET# File-Nr., Variable
- CMD File-Nr.
- CLOSE File-Nr.

File-Name

»File« wird normalerweise mit »Datei« übersetzt.

Einen Datei-Namen gibt es aber nur beim OPEN-Befehl. Bei den Befehlen LOAD, SAVE und VERIFY ist der Name des Programms gemeint, der bekanntlich in Gänsefüßchen hinter diesen Befehlen steht. Bei Disketten als Pflicht, bei Kassetten als Option. Beim OPEN-Befehl steht der Name zwar auch in Gänsefüßchen, aber nicht direkt hinter dem Befehl, sondern erst an vierter Stelle. Auch er kann bei Kassetten-Betrieb, aber auch beim Drucker weggelassen werden.

Bei Disketten-Dateien stehen hinter dem Namen - als sein Bestandteil - noch Typ und Modus. Typ bezeichnet die Art der Datei (RELative, SEQuential, USeR, PRoGramm), Modus die Operation (Read, Write, Append).

Der File-Name kann bei Disketten auch eine Anweisung an das Betriebssystem des Disketten-Laufwerks sein (NEW, SCRATCH, RENAME etc.). Nähere Erklärungen dazu finden Sie im Handbuch des Disketten-Laufwerks.

Da in 187/188 die Adresse angegeben ist, ab der der Programm- beziehungsweise Datei-Name im Speicher steht, können wir ihn dort ansehen.

Geben Sie direkt ein

```
LOAD "1234"
```

Nach der RETURN-Taste kommt die übliche Aufforderung der Datasette, die wir aber mit der STOP-Taste abwürgen. In 187/188 steht jetzt die Anfangsadresse »A« des gespeicherten Namens:

```
A = PEEK(187)+256*PEEK(188):PRINT PEEK(A)
```

Wir erhalten die Zahl 49, das ist der ASCII-Codewert der Ziffer »1«. Durch Erhöhen der Anfangsadresse »A« im PRINT-Befehl um jeweils 1 kommen auch die übrigen Zeichen des Namens zum Vorschein.

Die VC 20-Besitzer kennen sicher den Effekt, der entsteht, wenn beim SAVE-Befehl vor dem File-Namen, aber innerhalb der Gänsefüßchen, eine Farbtaste eingegeben wird. Die FOUND-Meldung druckt nämlich den File-Namen in der gewählten Farbe aus, was besonders bei Kassetten-Operationen sehr eindrucksvoll ist.

Das Disketten-Laufwerk bietet eine andere Überraschung. Die geSHIFTete SPACE-Taste innerhalb des File-Namens erzeugt das zweite Gänsefüßchen.

```
SAVE"SPIEL 2 (SHIFT-SPACE) SYS 16000",8
```

erscheint in der Directory als:

```
"SPIEL 2" SYS 16000
```

und kann mit LOAD"SPIEL 2",8 geladen werden.

Auf diese Weise können Sie in einer gut lesbaren Art Anweisungen in den Namen eines Programms einbauen. Vorsicht ist allerdings geboten, wenn Sie die SHIFT-SPACE-Taste vor dem File-Namen drücken. Dann wird nämlich der 2. Gänsefuß direkt nach dem ersten gesetzt, und der nachfolgende File-Name steht außerhalb der Gänsefüße. Ein derartiges Programm ist von Uneingeweihten nur sehr schwer zu laden.

Geräte-Nummer

Alle an den Computer anschließbaren Peripheriegeräte haben eine eigene Nummer, unter welcher sie »angesprochen« werden können. Es handelt sich eigentlich um eine Adresse, und in der Tat sprechen manche Autoren von der »Primär-Adresse«. In der Beschreibung der Speicherzelle 184 sind sie alle zusammengestellt.

Bei den Befehlen LOAD, SAVE, VERIFY und OPEN steht diese Geräte-Nummer als zweite Angabe, vor der ersten Angabe durch ein Komma getrennt:

LOAD"NAME",1.....bedeutet "Laden von Datasette"
LOAD"NAME",8.....bedeutet "Laden von Diskette"
OPEN 12,4.....bedeutet "An den Drucker"

(Die 12 hinter dem OPEN-Befehl ist eine »File-Nummer«. Sie kommt weiter unten an die Reihe.)

Die Geräte-Nummer kann auch weggelassen werden. Dann allerdings nimmt der Computer automatisch an, daß es sich um die Datasette handelt. Die Geräte-Nummer steht jeweils in der Speicherzelle 186 und kann von dort mit PEEK ausgelesen werden.

Sekundär-Adresse

Die dritte Angabe hinter LOAD, SAVE und OPEN hat widersprüchliche Namen, wohl deshalb, weil sie abhängig von der Gerätenummer verschiedene Aufgaben hat. Von den ganz am Anfang schon genannten Varianten gefällt mir »Sekundär-Adresse« am besten.

Tabelle 6 faßt die Funktionen der Sekundär-Adresse zusammen.

Befehl	Sekundär-adresse	Funktion
LOAD	0	lädt Programm an den Anfang des Programmspeichers
	1	lädt ein Programm absolut, also an die Adresse, von der ab es gespeichert wurde.
SAVE	0	Normales SAVE, Programm wird bei späterem LOAD an den Anfang des Programm-Speichers geladen (Basic-Programme)
	1	erzwingt bei späterem LOAD des Programms die Speicherung ab der Adresse, wo es zur Zeit steht (Maschinen-Programme)
	2	setzt am Programmende auf der Kassette eine »Band-Ende«-Markierung, die beim »Überlesen« das Band mit der Fehlermeldung »DEVICE NOT PRESENT« stoppt.
	3	Kombination von 1 und 2
OPEN bei Kassette	0	Daten lesen
	1	Daten schreiben
	2	Daten schreiben mit »Bandende«-Markierung
OPEN bei Diskette	0	vom Betriebssystem der Floppy für »Laden« reserviert
bei Diskette	1	vom Betriebssystem der Floppy für »Speichern« reserviert
	2-14	reserviert numerierten Daten-Kanal, bis zu drei gleichzeitig
	15	reserviert Kommando-Kanal (nähere Angaben zu diesen Sekundär-Adressen siehe Floppy-Handbuch)
OPEN bei Drucker	0-10	die Funktionen sind bei den Druckern zum Teil verschieden. Bitte in Drucker-Anleitung nachsehen.

Tabelle 6. Funktionen der Sekundäradressen

Sie sehen, die Sekundär-Adressen haben es in sich! Die gerade benutzte Sekundär-Adresse steht in der Speicherzelle 185 und kann mit PEEK von dort ausgelesen werden.

File-Nummer

Die File-Nummer, oft auch logische File-Nummer genannt, steht als erste Angabe hinter dem OPEN-Befehl und den damit verbundenen PRINT#-, INPUT#-, GET#- und CLOSE-Befehlen.

Sie gibt einer zu bearbeitenden Datei eine Nummer, die von den nachfolgenden, anderen Befehlen ebenfalls verwendet werden muß, wenn sie sich auf dieselbe Datei beziehen. Auf diese Weise können mehrere Dateien nebeneinander bearbeitet werden, und zwar fünf bei Disketten und auf alle Geräte verteilt insgesamt zehn.

File-Nummern dürfen die Werte von 1 bis 255 haben. Bei Werten über 127 wird bei einem PRINT#-Befehl nach jedem RETURN-Zeichen - CHR$(13) - zusätzlich ein ASCII-Code für Zeilenvorschub - CHR$(10) - an das Gerät gegeben. Diese Eigenschaft kann bei denjenigen Geräten nützlich sein, die normalerweise auf CHR$(13) ohne Zeilenvorschub reagieren (kein automatisches Line-Feed).

Die File-Nummer steht jeweils in der Speicherstelle 184, von wo sie mit PEEK ausgelesen werden kann. Zum Beispiel, um nachzuprüfen, welches Gerät als letztes angesprochen worden ist.

Abschließend möchte ich nochmals darauf hinweisen, daß in den Speicherzellen 183 bis 188 immer die gerade als letzte verwendete Angabe steht. Wir haben aber gesehen, daß der Computer sich maximal zehn File-Nummern mit dazugehörenden Geräte-Nummern, Sekundär-Adressen etc. merken kann.

Er tut dies in speziellen Tabellen, die in den Speicherzellen 601 bis 630 stehen.

Wir werden uns also noch einmal mit dieser Sache befassen, wenn wir bei den entsprechenden Adressen angekommen sind.

Adresse 183 ($B7)

Länge des derzeitigen File-Namens

Die LOAD-, SAVE- und VERIFY-Befehle für Disketten verlangen die Angabe eines Programm- oder Dateinamens, auf Computerdeutsch »File-Name«. Nähere Angaben dazu finden Sie im Texteinschub Nr. 19 »Files - Geräte - Namen - Nummern«.

Auch der OPEN-Befehl kann einen File-Namen haben. Bei Kassettenoperationen kann der File-Name weggelassen werden.

In der Speicherzelle 183 steht während und nach der Verwendung eines der oben genannten Befehle eine Zahl, die angibt, aus wie vielen Zeichen der File-Name besteht.

Bei Disketten sind File-Namen möglich, die aus maximal 16 Zeichen bestehen.

Bei Kassetten dagegen sind Namenslängen von maximal 187 Zeichen erlaubt. Allerdings werden vom Computer auf dem Bildschirm nur 16 Zeichen ausgedruckt (siehe dazu den Texteinschub 20 »Tape-Header«).

Für die Längenangabe in Zelle 183 gilt dabei nur die Anzahl derjenigen Zeichen, die zwischen den Gänsefüßchen stehen.

Diese Zahl kann nach einer Ein-/Ausgabeoperation, auch nach einer ungültigen oder abgebrochenen, durch PEEK (183) ausgelesen werden.

Ein File-Name wird übrigens auch bei einem OPEN-Befehl der RS232-Schnittstelle angegeben. Dieser Name, der bis zu vier Zeichen lang sein kann, wird in die Speicherzellen 659 bis 662 übertragen und gibt dort die Übertragungsrate, Wortlänge und Parity-Prüfung an.

Adresse 184 ($B8)

Nummer der derzeitigen Datei (File)

Hinter jedem OPEN-Befehl steht eine Zahl, die der durch diesen Befehl angefangenen Datei zugeordnet wird. Diese Datei- oder File-Nummer gilt als Referenz für alle anderen Ein- und Ausgabebefehle derselben Datei. Nähere Angaben dazu können Sie dem Texteinschub Nr. 19 »Files - Geräte - Namen - Nummern« entnehmen.

Ein OPEN-Befehl ruft die entsprechende Routine des Betriebssystems auf, welche die File-Nummer in die Speicherzelle 184 schreibt. Vor dort kann sie mit PEEK(184) ausgelesen werden. Geben Sie die folgende Zeile direkt ein:

```
A=30:OPEN A,3:PRINT PEEK
(184):CLOSE A
```

Um verschiedene File-Nummern auszuprobieren, definieren wir sie als Variable A. Nach dem »A« des OPEN-Befehls steht die Zahl 3. Damit wird der Bildschirm angewählt (siehe »Sekundär-Adresse« im schon erwähnten Texteinschub). Das Anwählen des Bildschirms vermeidet eine störende Meldung des Betriebssystems.

Mit RETURN nach der obenstehenden Zeile wird der jeweilige Wert von A als Inhalt der Zelle 184 ausgedruckt.

Adresse 185 ($B9)

Derzeitige Sekundär-Adresse

Die Sekundär-Adresse steht als dritte Angabe hinter den Ein- und Ausgabe-Befehlen LOAD, SAVE, VERIFY und OPEN. Sie hat bei den verschiedenen Peripheriegeräten spezielle Funktionen. Diese Funktionen sind im Texteinschub 19 näher erläutert.

Der jeweilige Wert der Sekundär-Adresse steht in der Speicherzelle 185, allerdings um 96 erhöht. Für Sekundär-Adressen stehen, über die Standardwerte der einzelnen Peripheriegeräte hinaus, die Zahlen von 0 bis 31 zur Verfügung. Ab 32 fängt in Zelle 185 wieder der Zyklus ab 0 an. Das können wir uns anschauen. Ich wähle zur Eröffnung einer Datei wieder den Bildschirm als »nicht-störendes« Empfangsgerät.

```
A=15:OPEN 1,3,A:PRINT
PEEK(185)-96:CLOSE 1
```

Durch Verändern des Wertes von A können Sie alle Möglichkeiten durchspielen.

Adresse 186 ($BA)

Derzeitige Geräte-Nummer

Jedes an den Computer anschließbare Gerät hat eine eigene Nummer, die zusammen mit den Ein-/Ausgabe-Befehlen LOAD, SAVE, VERIFY und OPEN angegeben werden muß. Wird keine Nummer angegeben, nimmt der Computer automatisch an, daß die Datasette gemeint ist.

Alle von Commodore vorgegebenen Geräte-Nummern sind in der folgenden Tabelle 5 aufgelistet.

Geräte-Nummer	angesprochenes Gerät
0	Tastatur
1	Datasette
2	RS232- (User-Port) Schnittstelle
3	Bildschirm
4	Drucker (normal)
5	Drucker (zusätzlich)
8	Disketten-Laufwerk Nr. 0
9	Disketten-Laufwerk Nr. 1
10, 11	weitere Disketten-Laufwerke

Tabelle 5. Von Commodore vorgegebene Geräte-Nummern

Die normale Geräte-Nummer eines Druckers ist 4, die eines Disketten-Laufwerks 8. Die zusätzlichen Nummern müssen gesondert am betreffenden Gerät eingestellt werden.

Nach der Ausführung eines der oben genannten Befehle steht die entsprechende Geräte-Nummer in der Speicherzelle 186, aus der sie mit PEEK(186) ausgelesen werden kann.

Adresse 187 und 188 ($BB und $BC)

Zeiger auf Adresse des derzeitigen File-Namens

Die Bedeutung eines Programm- oder Dateinamens - normalerweise kurz »File-Name« genannt - ist im Texteinschub Nr. 19 »File - Geräte - Namen - Nummern« näher beschrieben. In den Speicherzellen 187 und 188 steht in der Low-/High-Byte-Darstellung ein Zeiger auf diejenige Adresse im Programm-Speicher, wo dieser Name gespeichert ist.

Eine Ausnahme ist hier der OPEN-Befehl der RS232-Schnittstelle. Ihr File-Name wird in die Speicherzellen 659 bis 662 gebracht, wo er verschiedene Parameter dieser Schnittstelle steuert.

Adresse 189 ($BD)

Zwischenspeicher für RS232-Parity-Prüfung und für Kassettenoperationen

Die RS232-Routinen benutzen diese Speicherzellen als Zwischenspeicher für ein Prüf-Byte (Parity-Prüfung) bei der Ausgabe. Die Parity-Prüfung habe ich kurz im Texteinschub Nr. 18 erklärt.

Auch die Kassetten-Routinen bedienen sich dieser Speicherzelle. Sie verwenden sie als Zwischenspeicher für das gerade

gesendete oder empfangene Zeichen.

Adresse 190 ($BE)

Blockzähler für Kassetten-Ein-/Ausgabe

Das Betriebssystem des Computers schreibt bei SAVE ein Programm zweimal auf das Band der Datasette. Beim LOAD-Befehl wird der erste Block in den Arbeitsspeicher des Computers geladen; der zweite - identische - Block wird dann mit dem ersten Block Byte für Byte verglichen, um Datenfehler auf dem nicht immer ganz zuverlässigen Bandmaterial zu erkennen.

In der Speicherzelle 190 wird dem Betriebssystem angezeigt, wie viele Blockteile bei diesem Prozeß noch gelesen oder gespeichert werden müssen. Vom Basic-Programm aus ist diese Speicherzelle nicht zugänglich.

Adresse 191 ($BF)

Zwischenspeicher für LOAD-Operationen vom Band

Diese Speicherzelle wird beim Laden eines Programms vom Band dazu benutzt, um Zeichen aus einzelnen Bits zusammenzusetzen.

Adresse 192 ($C0)

Motorsperre der Datasette

Die Tasten der Datasette werden 60mal in der Sekunde von der »Interrupt-Routine« des Betriebssystem überprüft, ob eine von ihnen gedrückt worden ist. Die Speicherzelle 192 spielt dabei eine entscheidende Rolle, beim C 64 allerdings in einer anderen Weise als beim VC 20. Wie sie diese Rolle beim C 64 spielt, ist im Zusammenhang mit der Speicherstelle 1 ganz am Anfang dieses Kurses beschrieben worden. Ich habe dabei in zwei Beispielen gezeigt, wie durch Abfrage des vierten Bits von Adresse 1 geprüft werden kann, ob eine Taste der Datasette gedrückt ist und wie der Motor durch Setzen und Löschen des Bit 5 der Zelle 1 ein- und ausgeschaltet werden kann. Vorausgesetzt, der Inhalt der Speicherzelle 192 ist ungleich Null und eine Taste der Datasette ist gedrückt.

Nun will ich, wie versprochen, denselben Vorgang für den VC 20 beschreiben.

Wie Sie sich vielleicht noch erinnern, wird die Speicherzelle 1 beim VC 20 nicht für die Steuerung der Ein- und Ausgänge des Mikroprozessors verwendet. Diese Rolle wird beim VC 20 durch zwei Register des »Versatile Interface Adapter« (VIA 6522-A) ausgefüllt.

Für die Abfrage der Datasetten-Tasten ist das sechste Bit des VIA-Registers 37151 zuständig. Bei gedrückter Taste steht es auf 1, sonst auf 0. Ein kleines Programm zeigt es Ihnen:

```
10  X = PEEK(37151)
20  PRINT X
30  IF X = 62 THEN 50
40  GOTO 10
50  PRINT"TASTE
    GEDRÜCKT"
```

Wenn keine Taste gedrückt ist, läuft ein Zahlenband mit 126 ab. Die entsprechende Darstellung als Dualzahl lautet 1111 1110. Bei einer gedrückten Taste steht in 37151 die Zahl 62, als Dualzahl 0011 1110. Wichtig, wie gesagt ist nur das zweithöchste Bit.

Mit der Abfrage der Zeile 30 springt beim Drücken einer Taste das Programm auf die Zeile 50 und druckt den Text aus.

Den Motor der Datasette können wir mit Hilfe des Registers 37184 schalten. Wie beim C 64 gilt auch jetzt, daß dazu die hier angesprochene Speicherzelle 192, auch Interlock-Register genannt, eine Zahl größer als 0 enthält und daß außerdem eine Taste der Datasette gedrückt ist. Drücken Sie auf PLAY und geben Sie direkt ein:

```
POKE 192,255
POKE 37148,251: DER MOTOR
BLEIBT STEHEN.
POKE 37148,252: DER MOTOR
LÄUFT LOS.
```

Bestimmend sind hier Bit 2, 3 und 4.

Zum Ausschalten muß lediglich Bit 2 auf 1 stehen, zum Einschalten die drei Bits auf 110. Jede Zahl, die als Dualzahl diese Bedingungen erfüllt, kann dafür hergenommen werden. Um unabhängig von den anderen Bits des Registers 37148 zu bleiben, die ja auch ganz bestimmte andere Funktionen haben, empfiehlt es sich, über Boole'sche Verknüpfungen nur die wichtigen drei Bits zu verändern. Die beiden POKE-Befehle sehen dann so aus:

Ausschalten:

```
POKE 37148,PEEK(37148) OR 2
```

Einschalten:

```
POKE 37148,PEEK(37148)
AND 12
```

Adresse 193 und 194 ($C1 und $C2)

Anfangsadresse für Ein-/Ausgabe-Operationen

In diesen Speicherzellen steht in Low-/High-Byte-Darstellung die Adresse, ab der ein Programm gerade geladen oder gespeichert wird. Diese Adresse wird übrigens von hier auch in die Speicherzellen 172 und 173 gebracht, die wir schon früher besprochen haben.

Bei LOAD und SAVE auf Band steht hier die Anfangsadresse des Bandpuffers (828). Im Bandpuffer steht allerdings nur der sogenannte Bandvorspann (auf englisch »Tape Header«), während der Hauptteil des Programms im Programmspeicher ab einer Adresse steht, auf die der Zeiger in den Speicherzellen 195 und 196 hinweist.

Adresse 195 und 196 ($C3 und $C4)

Zeiger auf den Anfang des Programms hinter dem Tape Header

Bei jedem LOAD- und SAVE-Befehl für Kassetten wird der Vorspann (Tape Header), in dem Programmtyp, Anfangs- und Endadresse aufgezeichnet sind, im Kassettenpuffer ab Adresse 828 gespeichert. Der eigentliche Teil des Programms steht dann im Programmspeicher.

In den Speicherzellen 195 und 196 steht in der Low-/High-Byte-Darstellung diese Adresse, ab der das Programm beginnt. Ich habe für alle diejenigen, die mit der Datasette arbeiten, im Texteinschub Nr. 20 »Tape-Header« die Zusammenhänge mit einem Beispiel dargestellt.

Adresse 197 ($C5)

Tasten-Code der zuletzt gedrückten Taste

Bei der Behandlung der Speicherzelle 145 habe ich Ihnen mit Wort und Bild beschrieben, wie die Tasten des Computers abgefragt werden. Die dabei für jede Taste entstehende Dualzahl wird in eine Dezimalzahl (0 bis 63) umgewandelt und zuerst in die Speicherzellen 203 beziehungsweise 653 gebracht. Zur Umwandlung und Abfrage der Zellen 203 und 653 bringe ich bei diesen Speicherzellen mehr Details. Nach der Prüfung, welche Taste gedrückt worden ist, wird die Codezahl von 203 in die Speicherzelle 197 gebracht und dort »aufgehoben«. Diese vermeintliche Verdoppelung wird vom Betriebssystem dafür gebraucht, um zu erkennen, ob die nächste gedrückte Taste mit der vorhergehenden identisch ist. Ist sie identisch, dann entscheidet der Inhalt der Speicherzelle 650, ob das Zeichen dieser Taste mehrfach ausgedruckt wird. In 650 steht die sogenannte Wiederholfunktion. Aber ich will nicht vorgreifen. Die Codezahlen der einzelnen Tasten werde ich bei der Besprechung der Zelle 203 auflisten.

Adresse 198 ($C6)

Anzahl der Zeichen im Tastaturpuffer

Die Funktion des Tastaturpuffers, zu dem wir bei den Speicherzellen 631 und 640 noch kommen werden, habe ich bereits in diesem Kurs, und zwar im Texteinschub Nr. 15 »Dynamische Tastenabfrage« erklärt. Dabei habe ich damals schon sozusagen im Vorgriff die Zelle 198 verwendet.

In dieser Speicherzelle steht die jeweilige Anzahl der Zeichen, die im Tastaturpuffer gespeichert sind und darauf warten, weiterverarbeitet zu werden.

Das folgende kleine Programm zeigt es.

```
10  GET A$
20  PRINT PEEK (198);A$
30  FOR J=1 3000:NEXT J
40  GOTO 10
```

Der GET-Befehl holt ein Zeichen aus dem Tastaturpuffer - sofern eines dort zu finden ist. Die Zeile 20 druckt die Anzahl der Zeichen im Puffer aus, daneben das erste dieser Zeichen. Dann folgt eine Warteschleife, die uns erlaubt, ganz schnell ein paar Tasten zu drücken. Danach springt das Programm an den Anfang zurück und arbeitet diese eingegebenen Zeichen ab. Es ist dabei deutlich zu sehen, wie durch den GET-Befehl bereits ein Zeichen aus dem Puffer genommen und dadurch der Inhalt der Zelle 198 sofort um 1 reduziert wird.

Der Inhalt der Speicherzelle 198 kann mit POKE auch verändert werden.

Eine sinnvolle Anwendung dieser Beeinflussung erlaubt der nicht gerade sehr populäre WAIT-Befehl.

Ersetzen Sie bitte im obigen Programm die Warteschleife der Zeile 30 durch:

```
30 POKE 198,0: WAIT 198,1
```

Zuerst wird dem Computer vorgegaukelt, daß der Tastaturpuffer leer sei. Durch den WAIT-Befehl wartet das Programm danach so lange, bis ein Zeichen im Tastaturpuffer erscheint und springt erst dann auf die nächste Zeile 40.

Wenn Sie nach dem WAIT-Befehl statt der 1 eine 2 eingeben, wartet diese Zeile entsprechend auf zwei Tasteneingaben. Allerdings wird in der Zeile 20 dann nur jedes zweite Zeichen ausgedruckt.

Adresse 199 ($C7)

Flagge für reverse Darstellung der Zeichen

Normalerweise steht in dieser Speicherzelle eine 0, was mit PRINT PEEK (199) leicht nachgeprüft werden kann.

Sobald in der Zelle 199 eine andere Zahl als 0 steht, werden alle Zeichen in der reversen Darstellung gedruckt. Das Betriebssystem des Computers erhöht nämlich in diesem Fall den jeweiligen Bildschirmcode der Zeichen um 128. Ein Blick in eine Tabelle der Bildschirmcodes bestätigt, daß die Codes aller reversen Zeichen um genau 128 höher sind als die der normalen Zeichen.

Den reversen Modus können wir bekanntlich direkt mit der Kombination der CTRL- und der RVS-ON-Taste oder aber mit PRINT CHR$(18) herstellen. Wenn Sie aber versuchen sollten, das direkt einzugeben, um dann wieder mit PRINT PEEK

Texteinschub Nr. 20
Tape Header

Wenn ein Programm oder eine Datei auf Band gespeichert wird, setzt der Computer vor das Programm einen Vorspann, der auf englisch »Tape Header« genannt wird. Da dieser Name weit verbreitet ist, will ich ihn hier beibehalten. Der Tape Header 192 Byte lang. Er enthält alle wichtigen Angaben über das nachfolgende Programm.

Beim Laden eines Programms wird der Tape Header im Kassettenpuffer gespeichert, für den die Speicherstellen 828 bis 1019 reserviert sind. Von dort kann der Inhalt des Tape Headers gelesen und analysiert werden.

Bevor wir das versuchen, will ich erst seine Zusammensetzung erklären.

Im **ersten Byte** steht eine Kennzahl für den Typ des Programms. Diese Kennzahl ist abhängig von der Sekundär-Adresse, die beim SAVEn eingegeben worden ist. Die Arten der Sekundär-Adressen und ihre Bedeutung sind im Texteinschub Nr. 19 »Files - Geräte - Namen - Nummern« genau beschrieben. Es gibt Kennzahlen von 1 bis 5.

In Anlehnung an die Erklärung der Sekundär-Adresse kann man die Kennzahl generell dadurch beschreiben, daß ein Programm mit Kennzahl 1 immer an den Anfang des zur Verfügung stehenden Programm-Speichers geladen wird. Hauptsächlich kommt das für Basic-Programme in Frage.

Eine Kennzahl 3 bewirkt, daß das Programm an diejenige Stelle des Programmspeichers geladen wird, wo es vor dem SAVEn gestanden hat. Das ist hauptsächlich der Fall bei Programmen in Maschinensprache.

In Verbindung mit der Bedeutung der Sekundär-Adresse kann man den Zusammenhang, wie Tabelle 7 zeigt, darstellen.

Sekundär-Adresse	Kennzahl	Bedeutung
0, leer gerade	1	Programm wird ab Anfang des Basic-Speichers geladen
	2	Basic-Programm Datenblock, gefolgt von 191 Datenbytes
ungerade	3	Programm wird ab Adresse geladen, die in 829/830 steht
	4	Kopf für Basic-Programm Datenblöcke, die mit GET # etc. eingelesen werden
	5	logisches Bandende, das Betriebssystem sucht keine weiteren Programme

Tabelle 7. Tape Header Kennziffern

In **Byte 2 und 3** steht in Low-/High-Darstellung die Adresse, ab der das Programm im Speicher des Computers stand, als es gespeichert wurde.

In **Byte 4 und 5** steht die entsprechende End-Adresse des Programms.

Ab **Byte 6 bis Byte 192** steht der Name des Programms. Er darf also maximal 187 Zeichen lang sein. Bei LOAD werden allerdings nur 16 Zeichen auf dem Bildschirm dargestellt.

Jetzt wollen wir das alles mit einem kleinen Experiment überprüfen.

Schreiben Sie bitte ein kleines Programm, es braucht nicht sehr sinnvoll zu sein, wie zum Beispiel:

```
10      REM TAPE HEADER
20      REM TEST PROGRAMM
```

Nehmen Sie ein leeres Band und laden das Programm mit einem Namen, der länger sein soll als 16 Zeichen, zum Beispiel:

```
SAVE "TEST PROGRAMM FUER INHALT TAPE HEADER"
```

Nach Drücken der RECORD- und PLAY-Tasten der Datasette meldet der Computer:

```
FOUND TEST PROGRAMM FU
```

Es werden also nur 16 Zeichen inklusive Leerzeichen gedruckt. Sobald das Programm geladen ist, schauen wir im Kassettenpuffer nach, was in den ersten fünf Byte steht, danach lesen wir die restlichen Byte des Puffers.

Geben Sie direkt, ohne Zeilennummer, ein:

```
FOR I=0 TO 4: PRINT PEEK(828+I);: NEXT
```

Sie erhalten die Zahlen 1 1 8 41 8 (beim VC 20 mit 3-K-Speichererweiterung 1 1 4 41 4)

Danach geben wir wiederum direkt ein:

```
FOR I=5 TO 192: PRINT CHR$(PEEK(828+I));: NEXT
```

Beim VC 20 geben Sie in der FOR...NEXT-Schleife eine kleine Zahl ein, da der Bildschirmspeicher beim VC 20 kleiner ist als beim C 64.

Jetzt erscheint der volle Programmname, gefolgt von nicht sichtbaren Leerstellen. Wenn Sie in der letzten Direkteingabe den CHR$-Teil weglassen, dann druckt die Zeile die ASCII-Codes aus, und Sie sehen dann die Leerstellen als Zahl 32.

Diese Resultate habe ich zur besseren Übersicht in Tabelle 8 dargestellt.

Adresse	828	829	830	831	832	833 etc.
Byte Nr.	1	2	3	4	5	6 etc.
Bedeutung	Kennzahl	Low-Byte	High-Byte	Low-Byte	High-Byte	Namen in ASCII-Code
Resultat	1	1	8	41	8	T etc.
bei C 64	1	(2089)		(2049)		
Resultat	1	1	4	41	4	T etc.
bei VC 20		(1065)		(1025)		

Tabelle 8. Tape Header Bytes

Die Kennzahl in Byte »1« können Sie dadurch verändern, daß Sie dem oben verwendeten SAVE-Direktbefehl nach dem langen Namen ein ,1,1 anhängen. Im Ausdruck steht dann die Kennzahl »3«.

Übrigens, wenn Sie in den Speicherzellen 195 und 196 nachschauen, finden Sie dort denselben Wert wie in den Zellen 829/830, so wie die Beschreibung es in der Memory Map erklärt.

Vielleicht fragen Sie jetzt nach dem Nutzen dieser ausführlichen Erklärung. Nun, hauptsächlich kann man damit Programme, die eigentlich wegen LOAD ERROR nicht mehr ladbar sind, retten. Oder aber man kann durch Verändern der Zahlen in den Bytes 2 bis 5 nachträglich die Adressen ändern, in die das Programm geladen wird. Die erste Anwendung werde ich erklären, sobald wir zu den Adressen des Kassetten-Puffers selbst kommen.

Das Problem des LOAD oder SAVE mit geänderten Adressen ist aber zu umfangreich für einen Texteinschub innerhalb dieses Kurses. Es wäre eigentlich einen eigenen kleinen Beitrag wert.

(199) nachzuschauen, was jetzt in der Speicherzelle 199 steht, dann werden Sie Schiffbruch erleiden. Das Betriebssystem setzt den Inhalt der Zelle 199 nach einem »Wagenrücklauf«, hervorgerufen zum Beispiel durch die RETURN-Taste oder nach einem PRINT-Befehl, der nicht mit einem Komma oder Semikolon abgeschlossen ist, sogleich auf 0 zurück. Natürlich erfolgt das auch durch Drücken der CTRL- und RVS-OFF-Taste.

Wir vermeiden die Rücksetzung durch einen Einzeiler:

```
PRINT CHR$(18) "AAA" PEEK
(199)
```

Wir erhalten drei reverse As und als Inhalt der Zelle 199 auch die Zahl 18. Dasselbe Ergebnis erhalten wir durch POKEn einer Zahl größer als 0 in die Zelle 199:

```
POKE 199,4: PRINT"XX" PEEK
(199)
```

Das Ergebnis beweist, daß diese Adresse sehr nützlich sein kann, zumal ihre Abfrage beziehungsweise Beeinflussung auch innerhalb eines Programms erfolgen kann.

Adresse 200 ($C8)

Zeiger auf das Ende der eingegebenen logischen Zeile

Eine echte Zeile faßt beim C 64 maximal 40 Zeichen, beim VC 20 nur 22.

Eine Zeile mit Anweisungen darf beim C 64 insgesamt 80 Zeichen, beim VC 20 sogar 88 Zeichen enthalten. Diese »verlängerte« Programmzeile nennt man »logische Zeile«.

Der Zeiger in Speicherzelle 200 gibt dem Betriebssystem an, auf welcher Position das letzte Zeichen einer eingegebenen logischen Zeile sitzt. Löschen Sie den Bildschirm und geben Sie direkt irgendwo auf dem Bildschirm den Befehl ein:

```
PRINT PEEK(200)
```

Sie erhalten die Zahl der Spalte des letzten Zeichens dieses Direkt-Befehls.

Adresse 201 und 202 ($C9 und $CA)

Zeiger auf Zeilen- und Spaltenposition des letzten Zeichens einer Zeile

Diese beiden Speicherzellen werden bei GET und INPUT verwendet, um die Zeile und Spalte des letzten Zeichens einer eingegebenen Zeile festzustellen. Die Spalten (in Zelle 201 angegeben) zählen von 1 bis 40 (1 bis 22 beim VC 20). Die Zeilen

```
2345678901234567890123456789012345678 90
0
1
1
2
2    PRINT PEEK(201) PEEK(202)
3 2  30
3
4
4
5
5
6      PRINT PEEK(201) PEEK(202)
6 6  32
7
7
8
8
```

Bild 20. Aufbau einer logischen Zeile beim C 64

```
 234567890123456789012
0
1
1
2
2
3
3  PRINTPEEK(201)
4 3
4
5
5
6
6
7    PRINTPEEK(202)
7  19
8
8
```

Bild 21. Aufbau einer logischen Zeile beim VC 20

(in Zelle 202 enthalten) zählen dagegen in Paaren von 0 bis 12, identisch mit der bei Zelle 200 erläuterten »logischen« Zweierzeile. Da dies nicht ganz einsichtig ist, gebe ich einen Bildschirmausschnitt wieder (Bild 20), der den Sachverhalt verdeutlichen soll.

Der erste Direktbefehl steht in der zweiten Zeile, das letzte Zeichen in der Spalte 30. Der zweite Befehl steht in der ersten Sechserzeile. Das heißt also, daß die Zeilenangabe dieselbe ist, egal, um welchen Teil der logischen Zeile es sich handelt. Das können Sie leicht nachprüfen, indem Sie den ersten Direktbefehl eine Zeile höher schreiben. Das Resultat ist dasselbe.

Die Unterscheidung, um welche der beiden Zeilenteile es sich handelt, wird in den Speicherzellen 217 bis 242 getroffen.

Beim VC 20 sieht der Bildschirmausdruck etwas anders aus (Bild 21), auch die Befehlseingabe habe ich der Zeilenlänge wegen verändert. Interessant ist beim VC 20 allerdings, daß dort trotz der Länge der logischen Zeile auch nur Zeilenpaare verwendet werden, deren Länge natürlich auf 22 Spalten reduziert ist.

Adresse 203 ($CB)

Tastencode der gerade gedrückten Taste

Bei der Speicherzelle 145 habe ich beschrieben, wie die Tasten des Computers abgefragt werden. Die dabei für jede der 64 Tasten (mit Ausnahme der RESTORE- und der SHIFT-LOCK-Tasten) entstehende Dualzahl wird in eine Dezimalzahl (0 bis 63) umgewandelt und in der Speicherzelle 203 gespeichert, einige auch in der Zelle 653. Diese Zahl steht auch in Speicherzelle 197, um sie mit der vorher gedrückten Taste vergleichen zu können.

Die Codezahlen jeder Taste lassen sich mit folgendem Programm abfragen:

```
10 PRINT PEEK (203)
20 GOTO 10
```

Nach RUN sehen wir ein laufendes Zahlenband, zuerst mit der Zahl 64. Das ist die Codezahl für »keine Taste gedrückt«. Die X-Taste ergibt 23 (26 beim VC 20), die W-Taste ergibt 9. Auch die Funktionstasten haben ihren Tastencode. F1 ergibt 4 (39 beim VC 20) und so weiter.

Nur die Steuertasten CTRL, SHIFT, und C= (Commodore-Taste) zeigen keine Reaktion. Deren Tastencode steht nämlich in Speicherzelle 653. Den Grund für diesen Separatismus erfahren Sie bei der Besprechung dieser Zelle. Hier ist nur interessant, daß nicht nur jede einzelne dieser drei Tasten einen eigenen Code hat, sondern auch alle machbaren Kombinationen von gleichzeitig gedrückten Steuertasten. Um das zu sehen, ändern Sie bitte die Zeile 10 so ab:

```
10 PRINT PEEK (203),
PEEK(653)
```

Tabelle 9 gibt Ihnen die volle Übersicht. Wenn Sie sich die Mühe machen, die Zahlenreihen der Zelle 203 auf Vollständigkeit zu prüfen, dann werden Sie feststellen, daß vier Zahlen fehlen. Es sind die Werte, die eigentlich den vier Steuertasten CTRL, C=, rechte und linke SHIFT-Taste zugewiesen sind. Aber wie gesagt, sie werden gleich nach 653 umgeleitet, wobei allerdings kein Unterschied mehr zwischen der linken und rechten SHIFT-Taste gemacht wird.

Einige Anwendungsbeispiele der Tastencodes sowie der Kombinationen der drei Steuertasten finden Sie im Texteinschub Nr. 21 »Abfrage der Tastencodes«. Wie schon erwähnt, haben die RESTORE-Taste und die SHIFT-LOCK-Taste keinen eigenen Code.

Die RESTORE-Taste ist überhaupt nicht an die Tastatur-Matrix angeschlossen, sondern ist direkt mit der RESTORE-Leitung des Computers verbunden. Dort löst sie einen sogenannten NMI-Interrupt aus. Die SHIFT-LOCK-Taste ist lediglich eine mechanische Verriegelung der SHIFT-Taste.

Adresse 204 ($CC)

Schalter für Cursor blinken

Ein Wert größer 0 in dieser Speicherzelle schaltet das Blinken des Cursors ab. Diese Abschaltung erfolgt durch das Betriebssystem immer dann, wenn sich Zeichen im Tastaturpuffer befinden und wenn ein Programm ausgeführt wird.

Im folgenden Beispiel einer Eingabe mit dem GET-Befehl, bei dem bekannterweise der Cursor nicht blinkt, wird demonstriert, daß durch POKE 204,0 der Cursor trotzdem blinkt. Das kann für selbstgeschriebene Eingabe-Routinen interessant sein.

```
10 PRINT"JA/NEIN?" ;
20 POKE 204,0
30 GET A$: IF A$=
"" THEN 30
40 PRINT A$
```

Umgekehrt kann man durch POKE 204,1 das Blinken des Cursors abschalten. Es bleibt dabei allerdings dem Zufall überlassen, ob er in der Ein- oder Aus-Phase abgeschaltet wird. Wenn Sie Pech haben, dann bleibt der Cursor bewegungslos stehen.

TASTE	C 64		VC 20	
	203	653	203	653
nichts	64	0	64	0
F1	4	0	39	0
F3	5	0	47	0
F5	6	0	55	0
F7	3	0	63	0
A	10	0	17	0
B	28	0	35	0
C	20	0	34	0
D	18	0	18	0
E	14	0	49	0
F	21	0	42	0
G	26	0	19	0
H	29	0	43	0
I	33	0	12	0
J	34	0	20	0
K	37	0	44	0
L	42	0	21	0
M	36	0	36	0
N	39	0	28	0
O	38	0	52	0
P	41	0	13	0
Q	62	0	48	0
R	17	0	10	0
S	13	0	41	0
T	22	0	50	0
U	30	0	51	0
V	31	0	27	0
W	9	0	9	0
X	23	0	26	0
Y	25	0	11	0
Z	12	0	33	0
1	56	0	0	0
2	59	0	56	0
3	8	0	1	0
4	11	0	57	0
5	16	0	2	0
6	19	0	58	0
7	24	0	3	0
8	27	0	59	0
9	32	0	4	0
0	35	0	60	0

TASTE	C 64		VC 20	
	203	653	203	653
+	40	0	5	0
-	43	0	61	0
*	49	0	14	0
/	55	0	30	0
=	53	0	46	0
↑	54	0	54	0
←	57	0	8	0
.	44	0	37	0
:	45	0	45	0
,	47	0	29	0
;	50	0	22	0
£	48	0	6	0
@	46	0	53	0
CRSR←	2	0	23	0
CRSR↑	7	0	31	0
DEL	0	0	7	0
HOME	51	0	62	0
STOP	63	0	24	0
RETURN	1	0	15	0
SPACE	60	0	32	0
SHIFT	64	1	64	1
C=	64	2	64	2
CTRL	64	4	64	4
SHIFT und C=	64	3	64	3
SHIFT und CTRL	64	5	64	5
C= und CTRL	64	6	64	6
SHIFT und C= und CTRL	64	7	64	7

Tabelle 9. Tabelle aller Tasten-Codes. Mit den beiden Speicherzellen (203 und 653) lassen sich insgesamt 476 Funktionstasten erzeugen

Adresse 205 ($CD)

Zähler für Blinkfrequenz des Cursors

Das Blinken des Cursors besorgt die Interrupt-Routine. 60mal in jeder Sekunde unterbricht sie den normalen Programmablauf. Während dieser Zeit führt sie mehrere »Haushalt«-Arbeiten durch. So wird hier die Tastatur abgefragt und das Cursorblinken gesteuert.

Dazu wird die Zahl 20 in die Speicherzelle 205 geschrieben und bei jeder Unterbrechung dann um 1 reduziert. Wenn die Zahl in 205 den Wert 0 erreicht hat, wird der Cursor eingeschaltet. Nach Adam Riese erfolgt das also 60/20 = 3mal pro Sekunde. Im Texteinschub Nr. 22 »Cursor-Spiele oder der INPUT-Befehl einmal etwas anders« wird mit diesem Zähler für die Blinkfrequenz experimentiert.

Adresse 206 ($CE)

Bildschirmcode des Zeichens unter dem Cursor

Im Prinzip ist der Cursor nichts anderes als das wiederholte Drucken eines Zeichens in reverser Form, das gerade unter dem Cursor steht. Normalerweise ist dies das Leerzeichen, deshalb sehen wir meistens das ausgefüllte Viereck. Fahren Sie aber mit dem Cursor auf einen Buchstaben, dann erscheint dieser wechselweise normal und revers. In Speicherzelle 206 steht jeweils der Bildschirmcode des Zeichens unter dem Cursor. Geben Sie die folgende Anweisung direkt ein, fahren aber noch vor dem Drücken der RETURN-Taste mit dem Cursor zurück auf eines der Zeichen, zum Beispiel auf ein P:

```
PRINT PEEK(206)
```

Nach RETURN erscheint die Zahl 16. Das ist also der Bildschirmcode des Zeichens, auf dem der Cursor saß, als die RETURN-Taste gedrückt wurde. Sie können das mit allen anderen Zeichen dieser Zeile wiederholen.

Ich kann mir vorstellen, daß eine derartige Abfrage in einem Programm, welches mit dem Bildschirm arbeitet, sinnvoll sein kann. Die Speicherzelle 206 wird allerdings nach jedem Blinken auf den neuesten Stand gebracht.

Texteinschub Nr. 21
Abfrage der Tastencodes oder 476 Funktionstasten

In der Speicherzelle 203 stehen die Tastencodes der gerade gedrückten Taste, insgesamt 64 an der Zahl. Vier davon, die Steuertasten CTRL, C= (Commodore-Taste), linke und rechte SHIFT-Taste, erscheinen allerdings dort nicht, sondern werden sofort in die Speicherzelle 653 umgeleitet. Dort erhalten sie (allerdings in mehrfacher Kombination) insgesamt acht Codewerte. Die Tabelle der Speicherzelle 203 zeigt alle Werte für den C 64 und den VC 20.

Anfänger der Computerei sitzen oft verzweifelt an dem Problem, die Funktionstasten der Commodore-Computer zum Leben zu erwecken. Nun, wir wissen, daß sie nur über die Abfrage ihrer Codewerte eingesetzt werden können.

Als Codewerte werden normalerweise nur die ASCII-Codes genannt.

Die schon erwähnte Tabelle zeigt jedoch, daß die Funktionstasten auch Tastencodes haben. Allerdings gibt uns das nur vier Möglichkeiten, entsprechend der Aufschrift für die ungeraden Funktionstasten-Zahlen. Um auch F2 bis F8 zu erhalten, drücken wir ja immer gleichzeitig die SHIFT-Taste. Das können wir bei der Abfrage der Tastencodes natürlich auch machen, indem wir uns den Inhalt der Zelle 203 und 653 ansehen. Das folgende kleine Programm überprüft, über den Tastaturcode, ob eine der acht Funktionstasten gedrückt wurde.

```
10 A=PEEK(203)
20 B=PEEK(653)
30 IF A=4 AND B=0 THEN PRINT"F1"
40 IF A=5 AND B=0 THEN PRINT"F3"
50 IF A=6 AND B=0 THEN PRINT"F5"
60 IF A=3 AND B=0 THEN PRINT"F7"
70 IF A=4 AND B=1 THEN PRINT"F2"
80 IF A=5 AND B=1 THEN PRINT"F4"
90 IF A=6 AND B=1 THEN PRINT"F6"
100 IF A=3 AND B=1 THEN PRINT"F8"
110 GOTO 10
```

Die Codezahlen gelten für den C 64, für den VC 20 müssen aus der Tabelle die entsprechenden Werte eingesetzt werden.

Wenn Sie sich anschauen, was in der Speicherzelle 653 alles passiert, dann werden Sie sicher sehen, wie willkürlich die Definition der geraden Funktionstasten ist. Statt der Kombination der Funktionstasten mit der SHIFT-Taste können wir genausogut die CTRL-Taste nehmen, oder die Commodore-Taste oder alle zwei oder...oder...!

Mit den acht Codewerten in Zelle 653 (0 bis 7) der acht möglichen Kombinationen der drei Steuertasten kann jede Funktionstaste acht Funktionen haben. Das ergibt insgesamt 32 Funktionstasten und nicht acht, wie die Aufschrift vermuten läßt. Einige davon werden in dem kleinen Demo(nstrations)-Programm eingesetzt. Zweck des Programms soll das Umschalten auf verschiedene Rahmen- und Hintergrundfarben sein. Für den C 64 gilt:

```
10 PRINT CHR$(147)
20 A=PEEK(203)
30 B=PEEK(653)
40 IF A=4 AND B=2 THEN POKE 53280,6:POKE 53281,7
50 IF A=5 AND B=2 THEN POKE 53280,5:POKE 53281,2
60 IF A=6 AND B=2 THEN POKE 53280,1:POKE 53281,1
70 IF A=1 AND B=7 THEN POKE 53280,3:POKE 53281,1
80 GOTO 20
```

Für den VC 20 gilt:

```
10 PRINT CHR$(147)
20 A=PEEK(203)
30 B=PEEK(653)
40 IF A=4 AND B=2 THEN POKE 36879,126
50 IF A=5 AND B=2 THEN POKE 36879,45
60 IF A=6 AND B=2 THEN POKE 36879,25
70 IF A=1 AND B=7 THEN POKE 36879,27
80 GOTO 20
```

Zeile 40 schaltet mit F1 und C= die Farbkombination BLAU/GELB ein.
Zeile 50 schaltet mit F3 und C= die Farbkombination ROT/GRÜN ein.
Zeile 60 schaltet mit F5 und C= die Farbe Weiß ein.

Als Spezialität schaltet Zeile 70 in den Normalzustand zurück, allerdings mit der seltenen Tastenkombination ← (Pfeil links) und alle drei Steuertasten (CTRL, SHIFT, C=) gleichzeitig gedrückt.

Jetzt aber kommt es noch ganz dick!

Ich habe oben gesagt, daß wir nicht acht, sondern 32 Funktionstasten haben. Die Verwendung der vier Funktionstasten in Kombination mit den acht Steuertastencodes in 653 macht es möglich. Dasselbe gilt natürlich für jede andere Taste auch! Zeile 70 im Demo-Programm beweist es.

Da uns insgesamt 60 Tasten zur Verfügung stehen, können wir theoretisch 480 Funktionstasten erzeugen - theoretisch, weil ja auch die STOP-Taste eine gültige Taste ist. Diese Taste steht uns allerdings nur in den Kombinationen mit der SHIFT-Taste zur Verfügung. Ohne SHIFT tut sie ihre Pflicht - sie stoppt. Mit SHIFT aber stoppt sie nicht, so daß wir insgesamt 472 mögliche Kombinationen haben - sicher mehr, als Sie je brauchen werden.

Übrigens, von den Kombinationen sind diejenigen mit der CTRL- oder Commodore-Taste in Spielen oder Anwenderprogramme wie Vizawrite oder Programmierhilfen sehr verbreitet. Ich kann Ihnen nur empfehlen, diese Art der Tastenabfrage ebenfalls zur Steuerung von Programm-Abläufen einzusetzen.

Texteinschub Nr. 22
Cursor-Spiele oder der Input-Befehl einmal etwas anders

Die Speicherzellen 204, 205 und 207 haben alle in einer bestimmten Weise mit dem Cursor zu tun. Da die Details bei jeder dieser Zellen behandelt worden sind, möchte ich hier zusammengefaßt ihren Einsatz an einem kleinen Demo-Programm zeigen. Die Idee zu diesem Programm stammt von Russ Davies (COMPUTE! Publications).

Russ Davies geht von der in vielen Leserbriefen geäußerten Unzufriedenheit mit dem INPUT-Befehl aus, der nicht beliebig lange Zeichenketten zuläßt und sich auch bei versehentlich gedrückter RETURN-Taste schlecht benimmt.

Eine Abhilfe wäre der GET-Befehl, aber der wiederum liefert keinen auffordernden Cursor. In diese Marktlücke springt das folgende kleine Programm, welches die prinzipiellen Anweisungen zeigt für:
- Eingabe langer Zeichenketten mit GET
- blinkender Cursor trotz GET
- veränderbares Blinken des Cursors

```
 10 POKE 211,0
 20 POKE 204,0:POKE 205,5
 30 FOR I=1 TO 40:NEXT
 40 GET A$
 50 IF A$=CHR$(13) THEN 100
 60 PRINT A$;
 70 X$=X$+A$
 80 GOTO 20
100 POKE 204,0:POKE 211,0
120 PRINT X$:PRINT:GOTO 20
```

Zeile 10 verwendet die Speicherzelle 211. Dieser Befehl, auch in Zeile 100, setzt den Cursor auf den Anfang der logischen Zeile zurück. Zeile 20 müßte eigentlich klar sein. Der Wert des POKE-Befehls für 205 ist interessant. Durch ihn kann die Blinkfrequenz des Cursors verändert werden. Bei diesem Programm ergibt der Wert 5 zusammen mit der Warteschleife in Zeile 30 eine mäßige Blinkfrequenz. Der Wert 1 läßt den Cursor eifrig zappeln.

Zeile 30, wie gesagt, dient zur Abstimmung der Cursorfrequenz, die von der Laufzeit der Programmschleife (20 bis 80) abhängt. Trotz des GET-Befehls in Zeile 40 blinkt der Cursor wegen der Flaggen in Speicherzellen 207 und 204.

Zeile 70 baut die Zeichenkette zusammen. Zeile 50 erlaubt ein Drücken der RETURN-Taste, wodurch lediglich die alte Zeichenkette mit der neuen Eingabe zusammengebunden wird. Einen Aussprung aus der Schleife will ich Ihnen selbst überlassen. Im vorliegenden Beispiel geht er nur über die STOP-Taste.

Adresse 207 ($CF)

Flagge für Blinkzustand des Cursors

In dieser Speicherzelle wird festgehalten, in welcher der beiden Blink-Phasen - normal oder revers - der Cursor sich gerade befindet. Eine 0 bedeutet reverses Zeichen, eine 1 bedeutet ein normales Zeichen.

Die Abfrage innerhalb eines Basic-Programms funktioniert nicht. Denn die Interrupt-Routine steuert den Phasenwechsel.

Adresse 208 ($D0)

Flagge für Eingabe von Tastatur oder Bildschirm

Diese Speicherzelle wird von einer Routine des Betriebssystems verwendet, die das jeweils nächste Zeichen in den Arbeitsspeicher holt. Für sie ist wichtig zu wissen, von welchem Eingabegerät dieses Zeichen geholt werden soll.

Wenn in der Zelle 208 eine 0 steht, wird damit die Tastatur als Eingabegerät bestimmt. Das ist der Normalfall, mit dem wir per Tastendruck Zeichen auf den Bildschirm tippen. Sobald aber statt einem Zeichen die RETURN-Taste gedrückt wird, ändert sich der Inhalt der Speicherzelle 208. Die oben genannte Routine überträgt nämlich jetzt den Inhalt der Zelle 213, in welcher die Länge der derzeitigen logischen Zeile steht, nach 208. Dann holt sie das nächste Zeichen, allerdings nicht von der Tastatur, sondern vom Bildschirm, und zwar das erste Zeichen der gerade abgeschlossenen logischen Zeile. Auf diese Weise gelangen die Anweisungen einer Zeile in den Arbeitsspeicher, wo sie im Direkt-Modus sofort ausgeführt, im Programm-Modus aber gespeichert und erst nach RUN ausgeführt werden.

Den Unterschied zwischen »logischer« und »echter« Zeile habe ich in dem Texteinschub Nr. 23 näher beschrieben.

Adresse 209 und 210 ($D1 und $D2)

Zeiger auf den Anfang der Bildschirmzeile, auf welcher der Cursor gerade steht

Dieser Zeiger in Low-/High-Byte-Darstellung zeigt auf die Adresse im Bildschirmspeicher, in welcher diejenige Zeile beginnt, auf der der Cursor gerade steht. Das läßt sich leicht nachprüfen durch folgende Programmzeile:

```
10 PRINT CHR$(147) PEEK(209)
PEEK(210)
```

Nach RUN wird erst der Bildschirm gelöscht, der Cursor in die HOME-Position gebracht und dann der Inhalt der beiden Zellen ausgedruckt. Da dies alles in der ersten Zeile passiert, sehen wir als Resultat eine 0 und eine 4. Die beiden Zahlen ergeben zusammen die Adresse, in der die erste Zeile des Bildschirmspeichers beginnt. Erweitern Sie die Zeile 10 um ein Komma und die Low-/High-Byte-Berechnung:

```
10 PRINT CHR$(147) PEEK(209)
PEEK(210), PEEK(209)+256*
PEEK(210)
```

Jetzt sehen wir als Resultat:
0 4 1024

Beim VC 20 erscheinen die der verwendeten Speichererweiterung entsprechenden Zahlen. Wir können durch einen TAB-Befehl den zweiten Teil der PRINT-Anweisung in die nächste Zeile schieben und sehen, was dann herauskommt:

```
20 PRINT PEEK(209) PEEK
(210),TAB(50) PEEK(209)+
256*PEEK(210)
```

Das Resultat ist jetzt:

```
0    4    1024
40   4    1104
```

Einen entsprechenden Zeiger für die Adresse der dazugehörigen Zeile im Farbspeicher werden wir in den Speicherzellen 243 und 244 antreffen. Durch POKEn können wir die Cursorposition leider nicht beeinflussen, aber Abfragen geht, wenn es uns interessiert.

Adresse 211 ($D3)

Position des Cursors innerhalb einer logischen Zeile

Den Inhalt der Speicherzelle 211 könnte man auch die Spaltenposition des Cursors nennen, wenn es sich nicht um die Position in der logischen Zeile handelte (siehe Texteinschub

Nr. 23). Beim C 64 sind daher die Werte von 0 bis 79, beim VC 20 von 0 bis 87 möglich.

Diese Speicherzelle zusammen mit Zelle 214 wird von den Befehlen POS, TAB, SPC und vom Komma innerhalb einer PRINT-Anweisung verwendet, um den Cursor zu positionieren. Das können wir auch. Um den Cursor auf Platz 5 in der Bildschirmzeile 18 zu bringen, geben wir folgende Programmzeile ein:

```
10 POKE 214,17:PRINT:POKE
211,5:PRINT"C 64"
```

Aus innerbetrieblichen Gründen muß der Wert, den wir als Zeile erzielen wollen, um 1 verringert in die Zelle 214 gePOKEt werden. Mit der Zahl 17 wird also der Cursor zuerst auf die Zeile 18 gebracht, dann in Spalte 5, ab der dann das Wort »C 64« gedruckt wird. Auf diese Weise erhalten wir einen Befehl, der in anderen Basic-Formen unter dem Namen PRINT AT sehr verbreitet ist, der bei den kleinen Commodore-Computern aber fehlt.

Der Vorgang dabei besteht darin, daß die Inhalte von 211 und 214 in das X-Register beziehungsweise in das Y-Register des Mikroprozessors gebracht werden. Von dort können die Werte dann von einer Routine des Betriebssystems abgerufen werden. Das klingt alles sehr nach Maschinensprache. Aber wir haben Glück, denn sowohl die beiden Register als auch die besagte Routine sind von Basic aus ansprechbar. Das X-Register steht in Speicherzelle 781, das Y-Register in Speicherzelle 782, die Routine beginnt sowohl beim C 64 als auch beim VC 20 ab der Adresse 58634, wo wir sie mit dem SYS-Befehl starten können.

Für unser Beispiel sieht das dann so aus:

```
10 POKE 781,18:POKE 782,5:
SYS 58634:PRINT"C 64"
```

Wir erhalten dasselbe Ergebnis, nur mit dem Unterschied, daß die Zeile jetzt wirklich die Zeile 18 ist. Mit dieser Methode ist jetzt auch die Zeile 0 erreichbar.

Die Speicherzellen 781 und 782 bieten natürlich noch andere Anwendungen, auf die wir noch kommen werden.

Adresse 212 ($D4)

Flagge für Gänsefuß-Modus

Steht in dieser Speicherzelle eine 0, dann befindet sich der Computer im Gänsefuß-Modus, andere Zahlen bedeuten den Normal-Modus.

Selbst Anfängern ist der Gänsefuß-Modus sehr rasch geläufig, bietet er doch die Möglichkeit, Zeichen mit der PRINT-Anweisung auszudrucken. Genauso bekannt sind aber auch die Tücken der Gänsefüße. Die Cursor-Tasten reagieren nicht wie gewohnt. Auch die Farbumschaltung und andere Steuertasten zeigen nicht die übliche Wirkung, sondern drucken – allzu oft unerwartet – ein reverses Zeichen auf den Bildschirm.

Eingeschaltet wird der Gänsefuß-Modus durch Drücken der geSHIFTeten 2-Taste oder der geSHIFTeten INST/DEL-Taste. Abgeschaltet wird er nach jedem 2., 4., 6., also nach jeder geradzahligen Wiederholung der Gänsefuß-Taste innerhalb einer Zeile. Abgeschaltet wird er auch durch die RETURN-Taste. Das spezielle Verhalten der Steuertasten zwischen Gänsefüßen läßt sich für faszinierende Effekte ausnutzen.

Leider läßt sich der Inhalt der Speicherzelle 212 und damit der Status des Gänsefuß-Modus von Basic aus nicht beeinflussen. Doch in Maschinensprache unter Verwendung der Interrupt-Routine geht es, und einige Vorschläge zum Abschalten des Gänsefuß-Modus per Tastendruck sind schon veröffentlicht worden.

Adresse 213 ($D5)

Länge der Bildschirmzeile

Im Texteinschub 23 »Logische und echte Zeilen« ist der Unterschied zwischen den beiden Zeilentypen beschrieben.

Der Inhalt dieser Speicherzelle entscheidet, wann eine neue logische Zeile begonnen werden muß oder ob die laufende logische Zeile um eine weitere echte Zeile erweitert werden kann. Der Bildschirm-Editor verwendet diese Speicherzelle, um komplette logische Zeilen nach oben zu verschieben. Einige andere Routinen benutzen den Wert der Zelle bei der Rückwärtsüberprüfung einer Zeile, bei der die Endposition der Zeile bekannt sein muß. Schließlich bezieht noch die bereits behandelte Speicherzelle 200 ihren Wert von der Zelle 213.

Adresse 214 ($D6)

Nummer der echten Zeile, auf der sich der Cursor gerade befindet

Diese Speicherzelle ist zusammen mit der Speicherzelle 211 beschrieben.

Adresse 215 ($D7)

Zwischenspeicher für den ASCII-Codewert der zuletzt gedrückten Taste

Bei der Tastaturabfrage werden die Tastencodes (siehe Speicherzelle 203) in ASCII-Codewerte umgewandelt und in den Tastaturpuffer gebracht. Die Speicherzelle 215 dient dabei als Zwischenspeicher. Kassettenoperationen speichern hier auch Prüfsummen ab.

Adresse 216 ($D8)

Flagge für INSERT-Modus

Immer wenn die geSHIFTete INST/DEL-Taste gedrückt wird, um in einer Zeile Platz für ein einzufügendes Zeichen zu schaffen, wird der Inhalt der Speicherzelle 216 um 1 erhöht. Dann wird die Zeile ab dem Freiplatz nach rechts verschoben, der Inhalt der Speicherzelle 213 erhöht und schließlich der entsprechende Wert der Link-Tabelle für Bildschirmzeilen ab Speicherzelle 217 bis 242 verändert.

Bei jedem Tippen eines Zeichens in den freigewordenen Platz wird der Inhalt von 216 wieder um 1 reduziert, bis mit der 0 das Ende des INSERT-Modus angezeigt wird.

Adresse 217 bis 242 ($D9 bis $F2)

Link-Tabellen der Bildschirm-Zeilen

Diese 26 Speicherzellen enthalten Angaben für jede Zeile des Bildschirms. Jedes dieser Bytes hat zwei Funktionen.

Die ersten 4 Bit, also Bit 0 bis 3, geben an, in welchem Speicherblock, man sagt auch »page« dazu, das erste Byte der betreffenden Bildschirmzeile sich befindet. Diese Angabe wird zur Berechnung des Zeigers in der Speicherzelle 209 (siehe dort) verwendet. Sie ist in dieser Form notwendig, da der Bildschirmspeicher beim C 64 überall in den Arbeitsspeicher gelegt werden kann. Um die Position eines Zeichens oder besser gesagt eines Bytes davon im Bildschirmspeicher genau positionieren zu können, braucht das Betriebssystem noch die genaue Lage innerhalb des Speicherblocks. Das Low-Byte dieser Zahl steht in einer Tabelle ab Speicherzelle 60656 (60952 beim VC 20). Das High-Byte wird berechnet, und zwar durch Addition des Wertes der Speicherzelle 648 mit dem Wert der ersten 4 Bit in Tabelle 217 bis 242. Der Wert in Zelle 648 gibt die Anfangsadresse des Bildschirmspeichers an.

Der zweite Teil jedes Bytes in der Tabelle 217 bis 242 hat eine andere Funktion. Wie im Texteinschub 23 beschrieben ist, kann eine logische Zeile aus ein oder zwei (beim VC 20 sogar bis zu 4) echten Zeilen bestehen. Das Betriebssystem braucht daher eine Angabe, welche echten Zeilen zu einer logischen Zeile verbunden sind. Dieses Verbinden heißt auf englisch »link«, daher heißt der Speicherbereich 217 bis 242 »Link-Tabelle«. Diese oberen 4 Bit zeigen mit irgendeinem Wert über 0 an, daß die betreffende echte Zeile die erste oder einzige einer logischen Zeile ist. Sind die 4 Bit alle 0, dann ist sie eine 2., 3. und 4. Zeile der logischen Zeile.

Adresse 243 und 244 ($F3 und $F4)

Zeiger auf Position des Cursors im Farbspeicher

Jedem Platz im Bildschirmspeicher, in dem der Codewert für ein Zeichen steht, entspricht ein Platz im Farbspeicher, in dem der Codewert für die Farbe dieses Zeichens steht.

Das heißt, daß den Bildschirm-Werten der Speicherzellen 209 bis 210 die Farbspeicher-Werte der Zellen 243 bis 244 entsprechen. Dieser Zeiger bestimmt also in der Low-/High-Byte-Darstellung die Adresse im Farbspeicher, ab der die echte Zeile beginnt, auf welcher der Cursor gerade steht.

Adresse 245 und 246 ($F5 und $F6)

Vektor auf die Decodiertabelle für ASCII-Codewerte der Tasten

Bei der Diskussion der Speicherzelle 145 habe ich Ihnen gezeigt, wie das Drücken einer der 64 Tasten entschlüsselt wird.

Ein entschlüsselter Wert wird in Speicherzelle 145 zwischengespeichert und gelangt dann als Tastencode in die Speicherzelle 203. Bei der Besprechung

der Zelle 203 wurden die Codewerte aufgelistet. Ich habe auch darauf hingewiesen, daß die Codes der drei Steuertasten SHIFT, CTRL und COMMODORE (C=) separat in der Zelle 653 stehen.

Diese Tastencodes sind sehr nützlich und vom Basic aus gut verwendbar. Im Verkehr mit anderen Geräten sind sie aber nicht einsetzbar, da sie keiner internationalen Norm entsprechen.

Eine derartige Norm bietet der sogenannte ASCII-Code. Deshalb rechnet, wo notwendig, das Betriebssystem die Tastencodes in den ASCII-Code um.

Dazu stehen im Speicher des Betriebssystems vier Tabellen (Bild 22), die die ASCII-Codewerte enthalten (in Klammern für den VC 20).

ab	60289	(60510):	normale Zeichen
ab	60354	(60575):	Zeichen mit SHIFT
ab	60419	(60640):	Zeichen mit C=
ab	60536	(60835):	Zeichen mit CTRL

Bild 22. Anfangsadressen der 4 Tabellen

Die Umrechnung der Tastencodes in ASCII-Code ist sehr einfach. Der Tastencode wird lediglich zu der Anfangsadresse der entsprechenden Tabelle hinzugezählt. Die Summe ergibt die Adresse in der Tabelle, in der der ASCII-Code für das gedrückte Zeichen steht.

Als Beispiel nehmen wir das normale »G«, sein Tastencode ist 26 (VC 20: 19). Zur Anfangsadresse der normalen Tabelle 60289 (60510) dazugezählt, ergibt das 60315 (60529). Schauen wir in dieser Speicherzelle nach:

```
PRINT PEEK (60315): REM
BEIM C 64
PRINT PEEK (60529): REM
BEIM VC 20
```

In beiden Fällen erhalten wir die Zahl 71. Ein Blick in die ASCII-Tabelle des Handbuchs bestätigt die Richtigkeit.

Der Vektor in den vorliegenden Speicherzellen 245/246 zeigt auf den Anfang der vier Tabellen, und zwar in Abhängigkeit davon, ob und welche der drei Steuertasten zusammen mit einer anderen Taste gedrückt worden ist. Auch das kann ich Ihnen zeigen mit einer Programmzeile, welche ein Zahlenband erzeugt, dessen Zahl durch die Steuertasten verändert wird. Sie werden sehen, es sind die Anfangsadressen der vier Tabellen.

```
10 PRINT PEEK(245)+256*
PEEK(246):GOTO 10
```

Adresse 247 und 248 ($F7 und $F8)

Zeiger auf den Anfang des RS232-Eingabe-Puffers

Immer wenn ein Kanal mit der Geräte-Nummer 2 (User-Port) eröffnet wird, werden am oberen Ende des Arbeitsspeichers zwei Pufferspeicher mit je 256 Byte reserviert (siehe auch die Beschreibung der Speicherzellen 55 bis 56).

Der Zeiger, der in Low-/High-Byte-Darstellung in 247 und 248 steht, zeigt auf die Anfangsadresse desjenigen Pufferspeichers, der die ankommenden Zeichen aufnimmt.

Ein Programm, das den User-Port benutzen will, sollte übrigens immer zuerst die Gerätenummer 2 eröffnen, bevor irgendwelche Variable definiert werden. Dadurch wird vermieden, daß die Puffer-Reservierung eventuelle Variablenwerte überschreibt, die bereits in diesen 512 Byte angesiedelt worden sind.

Adresse 249 und 250 ($F9 und $FA)

Zeiger auf den Anfang des RS232-Ausgabe-Puffers

Dieser Zeiger ist der Zwilling zu dem in den Zellen 247/248 stehenden Zeiger, diesmal aber für den Ausgabe-Puffer.

Adresse 251 bis 254 ($FB bis $FE)

Vier freie Byte für Anwenderprogramme

Diese 4 Byte sind frei, und da sie von Basic nicht gestört beziehungsweise verändert werden, eignen sie sich in idealer Weise für Flaggen, Register oder andere Zwischenspeicher.

Adresse 255 ($FF)

Zwischenspeicher für Daten bei der Umwandlung von Gleitkomma-Zahlen in ASCII-Werte

Der Titel dieser Speicherzelle sagt schon alles.

Jetzt haben wir eine erste Etappe unserer Wanderung durch die Speicherlandschaft hinter uns, nämlich den Bereich von 0 bis 255. Man nennt ihn »Zero-Page«, und er hat in der Maschinensprache-Programmierung beziehungsweise in der Adressierung eine spezielle Bedeutung. Für Basic-Programmierer ist diese Seiteneinteilung bedeutungslos, und wir werden deshalb unbeirrt als nächstes mit einem größeren Speicherblock, nämlich von 256 bis 511 weitermachen.

Texteinschub Nr. 23
Logische und echte Zeilen

Der Bildschirm des C 64 besteht aus 25 Zeilen. Jede davon enthält 40 Stellen. Der VC 20 hat 23 Zeilen mit je 22 Stellen. Diese Zeilen des Bildschirms werden »echte Zeilen« genannt.

Der Begriff »Zeile« kommt auch beim Programmieren vor. Dort bedeutet er die Reihe der Anweisungen und Befehle, die hinter einer Zeilennummer zusammengefaßt sind. Diese Programmzeilen werden »logische Zeilen« genannt.

Wozu dient dieser Unterschied?

Der Grund ist ganz einfach der, daß sehr oft eine Programmzeile mehr Zeichen enthält, als in eine (echte) Zeile des Bildschirms hineinpassen. Besonders Texte innerhalb einer PRINT-Anweisung sind häufig viel länger als 40 Zeichen, erst recht als die 22 Zeichen beim VC 20.

Deshalb läßt es das Betriebssystem des Computers zu, daß in einer (logischen) Programmzeile mehr als 40 (22) Zeichen stehen.

Beim C 64 kann eine logische Zeile aus zwei echten Zeilen bestehen mit einer maximalen Zeichenzahl von 80.

Der VC 20 erlaubt in einer logischen Zeile maximal 88 Zeichen und braucht dazu 4 echte Zeilen.

Dieses Zusammenfügen von mehreren echten Zeilen zu einer logischen Zeile hat natürlich Konsequenzen. Am meisten davon betroffen ist der Editor, das ist der Teil des Betriebssystems, der uns erlaubt, auf dem Bildschirm zu arbeiten, zu ändern, zu löschen und fertige Zeilen mit RETURN einzugeben.

Ich will Ihnen nur ein paar Beispiele nennen, die Sie sicher kennen, um Ihnen den Zusammenhang zu zeigen.

Wenn Sie ein Programm LISTen und anschließend in einer Programmzeile weitere Befehle anhängen, die über die Länge der echten Zeile hinausgehen, brauchen Sie selbst keinen Platz dafür schaffen. Das macht der Editor automatisch, indem er alle folgenden Zeilen auf dem Bildschirm nach unten schiebt. Für die Einsteiger unter Ihnen führe ich das vor. Geben Sie bitte die folgenden drei Zeilen ein (mit RETURN abschließen):

```
10 PRINT "A"
20 PRINT "B"
30 PRINT "C"
```

Jetzt fahren Sie mit dem Cursor auf die Zeile 20 und schreiben statt einem B so viele davon, bis Sie damit in die nächste Zeile kommen.

Sie werden sehen, daß dabei die Zeile 30 von selbst eins weiter nach unten rutscht. Erst wenn Sie in die 3. Zeile (5. Zeile beim VC 20) kommen, wird die Zeile 30 überschrieben. Sie sind dann über das Ende der logischen Zeile geraten. Eine überfüllte logische Zeile kann nicht eingegeben werden, die RETURN-Taste wird mit SYNTAX ERROR quittiert beziehungsweise abgewiesen.

Es gibt allerdings eine Methode, wie Sie in einem Listing eines Programms Programmzeilen, also logische Zeilen mit mehr als 80 (88) Zeichen erhalten können.

Fast alle Basic-Befehle können in abgekürzter Form eingetippt werden, am häufigsten wird sicher das Fragezeichen ? anstelle von PRINT eingesetzt. Sie können nun eine logische Zeile mit Abkürzungen und durch Weglassen von allen Zwischenräumen bis zum erlaubten Maximum füllen. Natürlich wird diese volle Zeile nach RETURN akzeptiert. Beim ausLISTen aber schreibt der Editor alle Befehle, auch die abgekürzten, in voller Länge aus und fügt nach der Zeilennummer einen Zwischenraum ein. Eine solche Zeile hat dann mehr echte Zeilen als erlaubt, intern aber hat sie die richtige Länge.

Einen Nachteil hat diese Komprimierung doch. In einer solchen »überlangen« logischen Zeile kann nicht mehr korrigiert

werden, es sei denn, sie wird vorher auf eine erlaubte Länge reduziert.

Abschließend möchte ich noch kurz erwähnen, daß zur Steuerung dieser Zusammenhänge zwischen echten und logischen Zeilen die Speicherzellen 200, 201, 202, 209, 211, 213, 214, 217 bis 242 und 658 eine entsprechende Rolle spielen, die im einzelnen in der Memory Map beschrieben ist.

Adresse 256 bis 266 ($100 bis $10A)

Arbeitsspeicher für Umwandlung von Gleitkomma-Zahlen in ASCII-Werte, auch FAC (Fließkomma-Akku) genannt

Diese 11 Byte werden von einer Routine des Betriebssystems verwendet, um Werte zwischenzuspeichern, die bei der Umwandlung von Gleitkomma-Zahlen in ASCII-Werte oder in Werte der Funktion TI$ anfallen. Eine andere Routine verwendet den Bereich, um Zeichenketten (Strings) zu untersuchen.

Adresse 256 bis 318 ($100 bis $13E)

Arbeitsspeicher für Fehler bei der Eingabe vom Band

Alle Daten, die auf Band gespeichert werden, stehen dort doppelt in zwei identischen Blöcken hintereinander. Beim Laden in den Computer werden beide Blöcke miteinander verglichen, um Fehler zu finden und, wo möglich, sie zu korrigieren.

In diesem Bereich, der übrigens auch bei der Speicherzelle 256 anfängt, aber 63 Byte in Anspruch nimmt, werden beim Laden Angaben gespeichert, aus denen das Betriebssystem erkennen kann, welche Bytes fehlerhaft sind.

Adresse 319 bis 511 ($13F bis $1FF)

Stapelspeicher (Stack) des Mikroprozessors

Die Funktionsweise eines Stapelspeichers, auf englisch »stack«, ist im Texteinschub Nr. 24 erklärt.

Der Stapelspeicher hat prinzipiell die Aufgabe, bei allen Sprüngen oder Unterbrechungen innerhalb eines normalen Programmablaufs alle Adressen und Daten so zu speichern, daß am Ende der Unterbrechung das Programm wieder fortgesetzt werden kann.

Derartige Unterbrechungen und Sprünge treten in Basic bei den Befehlen GOSUB-RETURN und FOR-NEXT auf, genauso wie bei vielen Routinen des Betriebssystems. In Maschinensprache gibt es dafür sogar eigene Befehle. Heimo Ponnath hat sie alle in seinem Assemblerkurs (Folge 7 und 8 im 64'er, Ausgabe 2/85 und 3/85) sehr ausführlich beschrieben.

Da uns hier Basic mehr interessiert, gebe ich Ihnen nur kurz an, was im Stapel gespeichert wird, da der Stapelspeicher nur in Maschinensprache manipuliert werden kann.

Jeder FOR-TO-NEXT-Befehl belegt 18 Byte im Stapelspeicher.

Im ersten Byte steht als Kennung die Zahl 129. Byte 2 und 3 enthalten in Low/High-Byte-Darstellung einen Zeiger auf die Adresse, in der die durch das FOR definierte Schleifen-Variable (zum Beispiel K in FOR K=0 TO 3) gespeichert ist. Die nächsten 5 Byte sind für den Gleitkommawert von STEP reserviert, das Byte danach für das Vorzeichen von STEP. Danach folgt der Gleitkommawert von TO mit 5 Byte und in zwei weiteren Byte die Nummer derjenigen Zeile, auf die nach dem NEXT zurückgesprungen wird. In den letzten beiden Bytes schließlich steht ein Zeiger auf der Adresse, in der das nächste Zeichen steht, welches nach Beendigung der FOR-TO-NEXT-Schleife gelesen werden muß.

Ein GOSUB-Befehl belegt 5 Byte im Stapelspeicher. Byte 1 enthält die Kennzahl 141. Ihr folgen zwei Byte für die Nummer der Zeile, auf die nach RETURN zurückgesprungen wird. Die letzten beiden Byte enthalten wieder einen Zeiger auf die Adresse, in der das nächste Zeichen steht, mit dem nach RETURN das Programm fortgesetzt wird.

Der Basic-Befehl DEF zur freien Definition von Funktionen belegt ebenfalls 5 Byte im Stapelspeicher. Ihre Verteilung ist dieselbe wie von GOSUB, mit dem einzigen Unterschied, daß statt der ersten Kennzahl irgendein anderer Wert verwendet wird, der aber keine Bedeutung hat.

Wenn so viele FOR...NEXT-Schleifen oder GOSUB-Sprünge gleichzeitig im Programm vorkommen, daß der Stapelspeicher voll wird, steigt das Programm mit OUT OF MEMORY aus.

Adresse 512 bis 600 ($200 bis $258)

Eingabespeicher von Basic

Wenn Sie Zeichen, zum Beispiel einen Befehl oder eine Programmzeile, eingeben und mit der RETURN-Taste abschließen, werden diese Zeichen in diesen Speicherbereich von 512 bis 600 gebracht. Seine Länge von 89 Byte entspricht der Länge einer logischen Zeile des VC 20 (88 Zeichen) plus einer Abschluß-Null. Die logische Zeilenlänge des C 64 von 80 Zeichen füllt den Speicherbereich nicht ganz aus, aber das Betriebssystem des C 64 ist gegenüber dem des VC 20 nicht geändert worden.

Nach RETURN sucht der Computer diesen Eingabespeicher nach Gänsefüßen, Komma und nach der Zahl für Zeilenende ab. Dann wandelt der Computer die gespeicherten Zeichen in für ihn lesbare Zahlen (Token und ASCII-Werte) um und fügt am Anhang die Zeilennummer und die Anschluß-Adresse (Link) der nächsten Zeile, am Ende die Abschluß-Null hinzu. Wenn eine Zeilennummer vorhanden ist, kommt alles in den Programmspeicher. Fehlt sie jedoch, dann wird die ganze Anweisung sofort ausgeführt (Direktmodus).

Eine detaillierte Beschreibung dieses Eingabe- und Umwandlungsvorganges gab Christoph Sauer im 3. Teil seines Kurses »Der gläserne VC 20« im 64'er, Ausgabe 11/84 ab Seite 126.

Dieser Speicherbereich wird auch von den Befehlen INPUT und GET benutzt, um die Eingabedaten aufzunehmen. Das erklärt übrigens, warum diese beiden Befehle nur innerhalb einer Programmzeile und nicht im Direktmodus verwendet werden können. Sie verwenden ja denselben Speicherplatz, der vom Direkt-(Eingabe-)Modus verwendet wird.

Es erklärt außerdem, warum eine von INPUT geforderte Eingabe maximal 88 Zeichen lang sein darf.

Adresse 601 bis 630 ($259 bis $276)

Tabellen für File-Nummern, Geräte-Nummern und Sekundär-Adressen von eröffneten Dateien

Bei der Besprechung der Speicherzelle 152 habe ich diesen Speicherbereich bereits erwähnt. Ich habe damals gesagt und gezeigt, daß die Zelle 152 über die Anzahl der eröffneten Dateien (Files) Buch führt, die Tabellen in 601 bis 630 dagegen darüber, welche File-Nummern, Geräte-Nummern und Sekundär-Adressen jeder eröffneten Datei zugeordnet sind. Wer sich nochmals über diese Begriffe orientieren will, den verweise ich auf die Beschreibung im Texteinschub Nr. 19.

Der Speicherbereich von 601 bis 630 ist in drei Blöcke unterteilt.

601 bis 610	Tabelle der File-Nummern
611 bis 620	Tabelle der Geräte-Nummern
621 bis 630	Tabelle der Sekundär-Adressen

In jeder Tabelle können also maximal 10 Byte stehen. Sie haben folgende Zusammensetzung:

Die drei Angaben über eine eröffnete Datei stehen in den Tabellen jeweils am gleichen Platz. Wenn also die Datei Nummer 5 als dritte Datei eröffnet worden ist, steht eine 5 in Zelle 603, ihre Gerätenummer in Zelle 613, die Sekundär-Adresse entsprechend in Zelle 623.

Immer wenn eine neue Datei eröffnet wird, kommen diese Angaben auf die nächsten Plätze der Tabellen, und der Inhalt der Speicherzelle 152 wird um 1 erhöht. Wird dagegen eine Datei geschlossen, dann rücken alle Angaben dahinter um eine Stelle zurück.

In diesen Tabellen kann nachgesehen werden, mit welchen Parametern Dateien eröffnet worden sind. Eine sehr interessante Anwendung, die Eintragung in den Tabellen durch POKE zu verändern, wurde von Rügheimer und Spanik veröffentlicht, welche ich hier zitieren möchte.

Eine Änderung der Filenummern in Tabelle 601 bis 610 ist nicht empfehlenswert, um Verwechslungen zu vermeiden.

Mit dem folgenden kleinen Programm, welches bei einer eröffneten Datei die Geräte-

Nummer in Tabelle 611 bis 620 ändert, kann zwischen einem Drucker mit Geräte-Nummer 4 und einem Plotter mit Geräte-Nummer 6 umgeschaltet werden.

```
10 OPEN 4,4,0
20 POKE 611,6: PRINT#4,
"PLOTTER"
30 POKE 611,4: PRINT#4,
"DRUCKER"
```

Ähnliches ist mit den Sekundär-Adressen möglich. Dabei muß man allerdings wissen, daß die Sekundär-Adressen nicht so wie sie sind in der Tabelle 621 bis 630 gespeichert werden, sondern mit »OR 96« verknüpft. Dasselbe müssen wir auch machen:

```
10 OPEN 3,4,0
20 POKE 621,0 OR 96:
PRINT#3, "GRAFIKMODUS"
30 POKE 621,7 OR 96:
PRINT#3, "TEXTMODUS"
```

Mit dieser Methode können Sie sich das Öffnen und Schließen vieler Dateien ersparen.

Adresse 631 bis 640 ($277 bis $280)

Tastaturpuffer

Bei der Behandlung der Speicherzelle 203 habe ich die Codezahlen beschrieben, die beim Drücken einer der 64 Tasten erzeugt werden. Bei den Speicherzellen 245 und 246 haben wir gesehen, wie aus diesen Tastencodes der ASCII-Code für die verschiedenen Zeichen einer Taste umgerechnet wird.

Hier nun im Tastaturpuffer landen diese umgerechneten ASCII-Werte. Wenn Sie den Kurs schon länger verfolgen, ist Ihnen das auch nicht neu, denn ich habe die Wirkungsweise des Tastaturpuffers bei der »Dynamischen Tastenabfrage« (Texteinschub Nr. 15) sozusagen im Vorgriff, ausführlich erklärt.

Zur Erinnerung sei gesagt, daß im Tastaturpuffer alle Zeichen zwischengespeichert werden, die während eines Programmlaufes eingegeben und nicht sofort vom Betriebssystem verarbeitet werden können. Sobald der Computer sich im Eingabe-Modus befindet — nach Programmende oder bei INPUT- und GET-Befehlen, werden die Zeichen in der Reihenfolge ihrer Eingabe herausgeholt und verwendet.

Der Tastaturpuffer ist 10 Byte lang. In Speicherzelle 198 steht, wieviele Zeichen sich im Puffer befinden.

Als Ergänzung zu den Beispielen der dynamischen Tastenabfrage zeige ich Ihnen im Texteinschub Nr. 25 »Programme, die sich selbst verändern« noch ein paar andere Anwendungen.

Adresse 641 und 642 ($281 und $282)

Zeiger auf den Anfang des Programmspeichers

Wenn der Computer eingeschaltet wird oder wenn mit einer Reset-Taste beziehungsweise mit SYS58260 (VC 20: SYS 58232) ein Kaltstart ausgelöst wird, setzt das Betriebssystem diesen Zeiger auf die Adresse des ersten freien RAM-Speicherplatzes.

Beim C 64 ist dies die Adresse 2048. Beim VC 20 hängt sie von der Speichererweiterung ab; ohne Erweiterung ist es 4096, mit einer 3-KByte-Erweiterung dagegen 1024, mit 8 KByte oder mehr ist die Adresse 4608.

Dieser Zeiger wird vom Basic-Übersetzer in die Speicherzelle 43 übernommen und nur von dort weiterverwendet.

Adresse 643 und 644 ($283 und $284)

Zeiger auf das Ende des Programmspeichers

Dieser Zeiger ist der Zwilling zu dem anderen Zeiger in 641 und 642. Er wird vom Betriebssystem auf die Adresse gesetzt, welche beim Kaltstart beziehungsweise der dabei durchgeführten Prüfung des Speichers den letzten verfügbaren RAM-Speicherplatz angibt. Beim C 64 ist diese Adresse normalerweise 40960 ($A000), beim VC 20 ohne Erweiterung 7680.

Dieser Zeiger wird vom Basic-Übersetzer in die Speicherzelle 55 übernommen.

Adresse 645 ($285)

Flagge für Ein- und Ausschalten der IEEE-488-Karte

Diese Speicherzelle ist etwas mysteriös. Sie kommt im ganzen Betriebssystem nur ein einziges Mal zum Einsatz, und zwar als Flagge beim Betrieb der sogenannten IEEE-488-Interface-Karte. Wenn diese Flagge gesetzt ist, wartet der Computer 64 Millisekunden lang, ob er von einem angeschlossenen Gerät angesprochen wird. Wenn kein Signal kommt, gibt er ein Fehlersignal aus.

Zahlen in der Zelle 645, die kleiner als 128 sind, bedeuten Flagge gesetzt, größer als 128 löschen sie die Flagge.

Texteinschub Nr. 24
Der Stapelspeicher

Der normale Speicher, mit dem wir es immer zu tun haben, in den wir Zahlen hineinPOKEn oder herausPEEKen, ist aufgebaut wie eine Häuserreihe, in der jedes Haus seine eigene Adresse hat. Wer etwas in einem bestimmten Haus abliefern oder aus ihm abholen will, muß seine genaue Adresse kennen.

Dieses Speicherprinzip heißt RAM, das ist die Abkürzung für »Random Access Memory« oder auf deutsch »Speicher mit wahlfreiem Zugriff«.

Der Stapelspeicher funktioniert anders. Jetzt werden die Häuser der Reihe nach aufgesucht, wie mit einer Postwurfsendung oder durch die Müllabfuhr. Ein besserer Vergleich ist der Aktenstapel. Die erste Akte wird auf den Tisch gelegt, alle nachfolgenden kommen obendrauf.

Beim Aktenstapel sieht man eine weitere Eigenschaft. Wenn er nämlich abgearbeitet wird, kommt die Akte als erste an die Reihe, die als letzte auf den Stapel gelegt worden ist.

Dieses Speicherprinzip heißt »LIFO«; das ist eine Abkürzung aus dem Englischen und heißt »Last In First Out«.

Der Benutzer eines Stapelspeichers braucht sich nicht mehr um Adressen zu kümmern. Er kennt nur noch den Platz, wo der Speicher Daten annimmt beziehungsweise abgibt. Was der Benutzer sich allerdings merken muß, ist die Reihenfolge seiner Ein- und Ausgaben.

Stapelspeicher werden von mehreren Programmiersprachen verwendet; am bekanntesten ist wohl FORTH, auch die HP-(Hewlett-Packard-)Taschenrechner arbeiten nach diesem Prinzip.

Unsere Computer verwenden das Prinzip des Stapelspeichers nur bei Programmschleifen, Unterprogramm-Sprüngen, ja überhaupt bei jeder Unterbrechung des normalen Programmablaufes. Das Problem dabei ist nämlich, sich alle Adressen und Angaben des Programms so zu merken, daß nach dem Ende der Unterbrechung das alte Programm lückenlos fortgesetzt werden kann.

Der Stapel belegt den Speicherbereich von 256 bis 511. Dieser Bereich unterscheidet sich in seinem Aufbau natürlich überhaupt nicht von den anderen Speicherbereichen. Es wäre auch viel zu aufwendig, alle Daten des Stapelspeichers bei jeder Ein- und Ausgabe rumzuschieben. Diese Aufgabe besorgt ein Register des Mikroprozessors, das »Stapelzeiger« genannt wird und das wie ein Zähler arbeitet.

Am Anfang steht in diesem Stapelzeiger die oberste Adresse 511. Bei jeder Eingabe wird die Zahl um 1 erniedrigt, bei jeder Ausgabe um 1 erhöht. Da der Stapelzeiger, wie jedes andere Register auch, eine Länge von 8 Bit hat, kann er nur die Dualzahlen von 00000000 bis 11111111 darstellen. Um daraus 256 beziehungsweise 511 zu formen, stellt der Mikroprozessor sozusagen fest verdrahtet ein neuntes Bit, immer auf 1, der Zahl voran.

Texteinschub Nr. 25
Programme, die sich selbst verändern

Der Speicherbereich von 631 bis 640 beherbergt den Tastaturpuffer. Schon im Texteinschub Nr. 15 habe ich Ihnen Anwendungen gezeigt, die den Tastaturpuffer einsetzen und in der Literatur unter dem Namen »Dynamische Tastenabfrage« bekannt sind.

Hinter diesem natürlich aus dem Englischen übersetzten Begriff verbirgt sich die Möglichkeit, innerhalb eines Programms bestimmte Werte in den Puffer zu speichern, die dort so lange aufgehoben bleiben, bis das Programm - aus welchen Gründen auch immer - beendet wird. Dann erst werden die Werte hervorgeholt. Wenn es Zeichen sind, dann erscheinen sie auf dem Bild-

schirm. Wenn es Steuertasten sind (zum Beispiel RETURN), werden sie ausgeführt.

Ich will hier nicht die schon gezeigten Anwendungen wiederholen, sondern sie lediglich um ein paar weitere Tricks ergänzen.

Zeile in ein Programm einfügen

Die folgende Methode ermöglicht die Veränderung eines Programms durch sich selbst. Genauer gesagt, man kann damit Programmzeilen einfügen. Nehmen wir an, Sie haben folgendes Teilprogramm:

```
150   PRINT"BITTE TASTE DRUECKEN"
160   GET A$:IF A$="" THEN 160
170   IF A$ <> "E" THEN 150
180   END
...
...
500   PRINT "NAECHSTER TEIL"
```

Die ersten vier Zeilen warten so lange, bis die E-Taste gedrückt wird, dann bleibt das Programm mit READY stehen.

Wir wollen nun eine zusätzliche Zeile 165 durch das Programm einfügen lassen, mit der nicht beendet, sondern durch Drücken der Q-Taste auf die Zeile 500 gesprungen wird. Das erreichen wir durch folgende Zeilen:

```
172   PRINT CHR$(147)
174   PRINT "165 IF A$=CHR$(81) THEN 500"
176   PRINT "GOTO 150"
178   POKE 631,19:POKE 632,13:POKE 633,13:
      POKE 634,13:POKE 198,4
```

Wenn jetzt die E-Taste gedrückt wird, löscht Zeile 172 den Bildschirm, Zeile 174 druckt in die zweite Bildschirmzeile die neue Programmzeile 165 und Zeile 176 darunter ohne Zeilennummer den Direktbefehl GOTO 150. In den Tastaturpuffer werden nacheinander die Werte für HOME und dreimal RETURN eingegeben und die Anzahl der Zeichen im Puffer auf 4 begrenzt.

Sobald nun das Programm den Befehl END in Zeile 180 erreicht, werden die Werte im Puffer ausgeführt, das heißt der Cursor geht nach HOME, das erste RETURN setzt ihn ohne Wirkung eine Zeile tiefer, wo er auf der neu ausgedruckten Zeile 165 steht. Das zweite RETURN gibt diese Zeile ein, das dritte RETURN führt den Direktbefehl GOTO 150 aus, wodurch das Programm weiterläuft, so wie vorher. Der einzige Unterschied ist nur, daß jetzt eine neue Zeile im Programm steht, nämlich die Zeile 165.

Prüfen Sie es mit STOP und LIST.

Zeile verändern oder löschen

Diese Methode habe ich von S. Leemon übernommen. Sie funktioniert im Prinzip genauso wie das Einfügen von Zeilen. Die Programmzeilen sind durch REMs erklärt. Probieren Sie es aus.

```
10    REM -- DIESE ZEILE WIRD GELÖSCHT
20    PRINT CHR$(147):PRINT:PRINT
30    PRINT "60 LIST"
40    PRINT "10"
50    PRINT "GOTO 70" CHR$(19)
60    FOR I=631 TO 633:POKE I,13:NEXT: POKE 198,3:END
70    REM -- DIESE ZEILE WIRD ERSETZT
```

Diese Verfahren, Zeilen eines Programms während des Laufs eines Programms zu ändern, haben sicher für einfache Programme keine große Bedeutung. Ich bin aber fest davon überzeugt, daß sie bei Programmen, die selbst lernen können, eine wichtige Rolle spielen. Nur habe ich hier jetzt noch kein gutes Demonstrationsbeispiel.

Einfügen einer Funktion

Sie alle kennen die Möglichkeit, Funktionen mit dem Basic-Befehl DEF FN.. selbst zu definieren. Vielleicht haben Sie auch schon einmal frustriert die Versuche abgebrochen, eine Funktion während des Programmablaufs per INPUT eingeben zu können. Das funktioniert nämlich nicht.

Mit dem Tastaturpuffer aber geht es, und zwar sehr elegant. Der vergebliche Versuch sieht so aus:

```
150   PRINT "FORMEL EINGEBEN"
160   INPUT "F(X)=";A$
...
250   DEF FNA(X)=A$
260   PRINT FNA(9)
270   GOTO 150
```

Der STRING A$, in Zeile 160 eingegeben, wird von Zeile 250 nicht übernommen. Zeile 250 können Sie so nicht verwenden, also löschen Sie sie bitte.

Sie können diese Zeile aber vom Programm einfügen lassen:

```
170   PRINT CHR$(147):PRINT:PRINT
180   PRINT "250 DEF FNA(X)=";A$
190   PRINT "GOTO 240"
200   PRINT CHR$(19)
210   POKE 631,13: POKE 632,13: POKE 198,2: END
...
...
240   REM -- FORMELBERECHNUNG --
```

Wenn Sie nicht genau nachvollziehen können, was da vorgeht, empfehle ich Ihnen, an das Ende der Zeile 200 noch ein END anzuhängen. Dann bleibt das Programm dort stehen, Sie sehen, wo der Cursor steht und können dann die Wirkung der Zeile 210 in Ruhe überlegen.

Eine weitere interessante Anwendung habe ich bei Russ Davies gefunden. Sie ermöglicht, Routinen des Betriebssystems oder Basic-Übersetzers aus dem Speicher auszulesen und als DATA-Anweisungen in Ihr Basic-Programm einzufügen.

Einfügen von DATA-Zeilen

Die Aufgabe besteht darin, aus einem auszuwählenden Speicherbereich - dort, wo die in Frage kommende Routine sitzt - den Maschinencode herauszuPEEKen und mit der inzwischen bekannten Methode in eine DATA-Zeile einzufügen. Ich gehe schrittweise vor:

```
20    PRINT CHR$(147): PRINT: PRINT
30    PRINT "5555 DATA";
50    FOR A=0 TO 3
60    PRINT PEEK(A)", ";:NEXT
70    PRINT CHR$(19)
80    POKE 631, 13: POKE 632,13: POKE 198,2: END
```

Mit Zeilen 20 und 30 wird der DATA-Zeile die Zeilennummer 5555 gegeben und in die 3. Bildschirmzeile geschrieben. Zeile 50 liest die Speicherzellen 0 bis 3 aus und druckt sie, mit einem Komma versehen, dahinter. Zeilen 70 und 80 geben diese Zeile in gewohnter Manier in das Programm ein.

Ganz richtig ist das noch nicht, da nach LIST die Zelle 5555 zu viele Zwischenräume hat:

```
5555 DATA 47 , 55 , 0 , 170 ,
```

Zur Korrektur müssen wir Zeile 60 erweitern:

```
60 PRINT MID$(STR$(PEEK(A)),2)",";:NEXT
```

STR$() wandelt den durch PEEK ausgelesenen Zahlenwert in einen String um. MID$(...,2) schneidet die Leerstellen weg.

Mr. Davies hat daraus eine komfortable kleine Routine gebaut, die den auszulesenden Speicherbereich und die gewünschte Zeilennummer der ersten DATA-Zeile abfragt, dafür sorgt, daß eine DATA-Zeile nicht länger als 16 Zeichen wird, in diesem Fall die nächste Zeilennummer um 10 erhöht und sogar das letzte Komma in jeder DATA-Zeile eliminiert.

Dieser ganze Komfort hat natürlich nichts mit dem Tastaturpuffer zu tun. Dieser Teil bleibt unverändert. Aber ich finde dieses Programm so durchdacht, daß ich es Ihnen als ein Beispiel guter Programmierung angeben will.

```
10    INPUT "STARTADRESE";A: INPUT "ENDADRESE";E:
      INPUT "ZEILE";Z
20    PRINT CHR$(147)
30    PRINT Z "DATA";
40    IF A> E THEN END
50    FOR A=A TO A+15 + (E<A+15)*(A+15-E)
60    PRINT MID$(STR$(PEEK(A)),2)",";:NEXT
70    PRINT"{CRSR-links}":PRINT "A="A":
      E="E": Z="Z+10": GOTO 20";
80    POKE 631, 19: POKE 632,13: POKE 633,13:
      POKE 634,13: POKE 198,4: END
```

Adresse 646 ($286)

Aktuelle Farbe der Zeichen (Vordergrundfarbe)

Um ein bestimmtes Zeichen auf den Bildschirm zu drucken, muß vom Betriebssystem erstens der Bildschirmcode des Zeichens in den Bildschirmspeicher und zweitens der Codewert der gewünschten Farbe in den Farbspeicher gebracht werden.

In der Speicherzelle 646 steht immer der Codewert derjenigen Farbe, die gerade eingestellt ist. Immer wenn ein PRINT-Befehl gegeben wird, holt das Betriebssystem den Farbwert aus der Zelle 646 und bringt ihn in den Farbspeicher, und zwar an den entsprechenden Platz, wo gerade gePRINTet werden soll. Der Codewert in der Zelle 646 kann auf drei Arten eingestellt werden:

- Drücken der CTRL-Taste gleichzeitig mit einer der Farbtasten 1 bis 8. Beim C 64 kommen noch weitere acht Farben dazu durch Drücken der Commodore-Taste anstelle der CTRL-Taste.
- PRINT-Befehl gefolgt vom ASCII-Codewert der Farbe innerhalb von Gänsefüßen.
- POKEn der Farbcodes 0 bis 7 (beim C 64 0 bis 15) direkt in die Speicherzelle.

Innerhalb eines Programms ist das POKEn in Zelle 646 wohl die eleganteste Methode (Tabelle 10).

Als Beispiel möge dieses kleine Programm dienen:

```
10 FOR X=0 TO 7
20 POKE 646,X
30 PRINT "A";
40 NEXT X
50 GOTO 10
```

Wer mehr über Vordergrund- und Hintergrundfarben erfahren will, der lese den Texteinschub Nr. 26 »Bunte Zeichen und bunter Hintergrund«.

FARBE	CODE	ASCII	TASTEN	FARBE	CODE	ASCII	TASTEN
schwarz	0	144	CTRL+1	orange	8	129	CBM+1
weiß	1	5	CTRL+2	braun	9	149	CBM+2
rot	2	28	CTRL+3	hellrot	10	150	CBM+3
lila	3	159	CTRL+4	dunkelgrau	11	151	CBM+4
purpur	4	156	CTRL+5	mittelgrau	12	152	CBM+5
grün	5	30	CTRL+6	hellgrün	13	153	CBM+6
blau	6	31	CTRL+7	hellblau	14	154	CBM+7
gelb	8	158	CTRL+8	hellgrau	15	155	CBM+8

Tabelle 10. Tabelle der Farben und ihrer Codes beziehungsweise Tasten

Adresse 647 ($287)

Zeichenfarbe unter dem Cursor

Das Blinken des Cursors wird dadurch erzeugt, daß das Zeichen auf der Stelle des Bildschirms, auf der er gerade steht (meistens ist es eine Leerstelle), dauernd von »normal« auf »revers« (oder »invertiert«) und zurück geschaltet wird. Die reverse Darstellung benutzt dabei die Farbe des Zeichens.

Genauso, wie sich der Computer in der Speicherzelle 206 das Zeichen merkt, mit dem er gerade blinkt, um beim Weiterwandern dieses Zeichen in seiner »normalen« Form auf dem Bildschirm zurückzulassen, merkt er sich die Farbe dieses Zeichens in der Speicherzelle 647.

Adresse 648 ($288)

Beginn des Bildschirmspeichers

In dieser Speicherzelle steht eine Zahl, die als High-Byte dem Betriebssystem angibt, ab welcher Speicherzelle der Bildschirmspeicher beginnt.

Nach einem Kaltstart (nach dem Einschalten oder nach dem Drücken der RESET-Taste) steht hier eine 4, das ergibt als Anfangsadresse 1024 (= 4*256). Beim VC 20 ohne Erweiterung steht dort eine 30. Daraus folgt, daß die Anfangsadresse bei 7680 (= 30*256) liegt.

Der Bildschirmspeicher hat keinen absolut festen Platz. Innerhalb gewisser Grenzen kann er durch Verändern des Inhalts der Speicherzelle 53272 (36869 beim VC 20) verschoben werden. Die Methode dazu ist im Texteinschub näher beschrieben. Wichtig dabei ist, daß nach dem Verschieben der Inhalt der Speicherzelle 648 entsprechend geändert wird, damit auch das Betriebssystem die Verschiebung berücksichtigt.

Umgekehrt kann aber dem Betriebssystem durch Ändern der Zahl in der Speicherzelle 648 mitgeteilt werden, daß es Zeichen in einen Speicherbereich bringen soll, der außerhalb des »offiziellen«, durch die Speicherzelle 53272 (36869) festgelegten Bildschirmspeichers liegt.

Zwei Beispiele sollen das verdeutlichen. Der PRINT-Befehl macht letztlich nichts anderes, als viele Zahlen in den Bildschirm- und den Farbspeicher zu POKEn. Wenn nun der Zeiger in Zelle 648 verschoben wird, kann man mit einem PRINT-Befehl eine beliebige Zeichenkette außerhalb des Bildschirmspeichers speichern. Auf die gleiche Weise kann man beim C 64 Sprites mit einem PRINT-Befehl speichern, ohne mit READ viele lästige DATA-Zeilen lesen zu müssen.

Adresse 649 ($289)

Maximale Länge des Tastaturpuffers

Der Tastaturpuffer belegt, wie schon besprochen, die Speicherzellen 631 bis 640. Er kann darin maximal 10 Zeichen zwischenspeichern.

Der Inhalt der Speicherzelle 649 legt fest, wieviele Zellen des Tastaturpuffers verwendet werden sollen, eine Zahl also, die normalerweise zwischen 0 und 10 liegen sollte. Die 10 ist übrigens der Wert, welcher nach dem Einschalten vom Betriebssystem in die Zelle 649 gebracht wird.

Diese Zahl wird immer mit dem Inhalt der Speicherzelle 198 verglichen, der die aktuelle Anzahl der Zeichen im Tastaturpuffer angibt. Ist die Differenz der beiden Zahlen gleich Null, dann können keine weiteren Zeichen eingegeben werden.

Es ist naheliegend, daß durch Verändern der Zahl in Zelle 649 die Länge des Tastaturpuffers verändert werden kann. Der eine Extremfall ist 0:
POKE 649,0 schaltet die Tastatur aus. Nichts geht mehr.

Das kann bei Programmen oder Spielen, die durch falsches oder zeitlich unpassendes Drücken von Tasten gestört werden, recht nützlich sein. Einschalten kann man dann die Tastatur nur mit RUN/STOP und RESTORE.

Auch eine Erhöhung der Zahl in 649 über 10 hinaus ist möglich. Die Zeichen werden halt nur über die dafür reservierten Speicherzellen 631 bis 640 hinaus in Zellen geschrieben, die eigentlich eine andere Funktion haben. Bis zur Speicherzelle 645 geht das normalerweise ohne Probleme, da die betroffenen »fremden« Adressen nur direkt nach dem Einschalten des Computers gebraucht werden.

Probieren Sie es aus, indem Sie zuerst eine Zeitschleife laufen lassen und in dieser Zeit etwa 20 Tasten drücken. Am Ende der Zeitschleife wird der Inhalt des Tastaturpuffers ausgedruckt, und Sie sehen in der Tat 15 der eingegebenen Zeichen:

```
POKE 649,15
FOR X=0 TO 10000:NEXT X
QWERTYUIOPASDFGHJKL
```

Auf dem Bildschirm erscheinen die Zeichen Q bis G.

Wenn Sie die Zahl in 649 noch weiter erhöhen, dringen Sie in die Zellen 646 und 647 ein und diese bestimmen bekanntlich die Zeichenfarbe. Wenn Sie aber eine unbeabsichtigte und unkontrollierbare Farbänderung nicht stört, können Sie den Tastaturpuffer auf 17 Zeichen vergrößern. Ab 18 Zeichen stürzt der Computer ab.

Adresse 650 ($28A)

Flagge für Tastenwiederholung

Normalerweise steht in dieser Speicherzelle eine 0. Das bedeutet, daß die Funktion der Cursor-Tasten, der Leertaste und der INST/DEL-Taste wiederholt wird, solange die entsprechende Taste gedrückt wird.

Durch Verändern der Zahl in der Speicherzelle 650 kann diese Wiederholfunktion sowohl auf alle Tasten ausgedehnt als auch für alle Tasten gesperrt werden.

```
POKE 650,0
```

ist der Normalzustand, Wiederholfunktion für Cursor-, Leer- und INST/DEL-Taste

```
POKE 650,64
```

schaltet Wiederholfunktion für alle Tasten aus.

```
POKE 650,128
```

erweitert Wiederholfunktion auf alle Tasten.

Adresse 651 ($28B)

Zähler für Wiederholgeschwindigkeit der Tasten

Das Betriebssystem verwendet diese Speicherzelle als Zähler, der die Geschwindigkeit bestimmt, mit der eine Taste wiederholt wird, wenn sie länger gedrückt wird. Voraussetzung ist die durch Zelle 650 festgelegte Wiederholbarkeit der Taste.

Am Anfang steht in der Zelle 651 die Zahl 6. Sobald eine wiederholbare Taste gedrückt wird, zählt das Betriebssystem diese Zahl alle 0,0167 Sekunden (60mal in der Sekunde) um 1 zurück, bis die Zahl 1 erreicht ist. Dann erst wird das Zeichen der gedrückten Taste wieder auf den Bildschirm gedruckt oder ihre Funktion wiederholt.

Bei jedem folgenden Lauf steht in Zelle 651 die Zahl 4. Entsprechend verkürzt sich der Zählvorgang.

Am schnellsten würde die Wiederholung natürlich mit dem Wert 1 in der Speicherzelle 651 sein. Von Basic aus mit POKE 651,1 geht das leider nicht.

Im Texteinschub Nr. 27 »Turbo-Tasten« wird ein Maschinenprogramm beschrieben, welches dies kann.

Adresse 652 ($28C)

Zähler für die Ansprechzeit der Wiederholfunktion von Tasten

Diese Speicherzelle wird vom Betriebssystem als Zähler verwendet, der festlegt, wie lange eine wiederholbare Taste gedrückt sein muß, bis die Wiederholfunktion einsetzt.

Am Anfang steht in der Zelle 652 die Zahl 16. Diese Zahl wird alle 0,0167 Sekunden um 1 reduziert, bis die Zahl 0 erreicht ist. Dann wird das Zeichen der Taste auf den Bildschirm gebracht oder ihre Funktion wiederholt. Anschließend wird die Zahl 4 in die Speicherzelle 651 geschrieben (siehe dort), während die Zelle 652 so lange auf 0 stehen bleibt, bis eine andere Taste gedrückt wird. Wie diese anfängliche Verzögerung reduziert werden kann, steht im Texteinschub Nr. 27 »Turbo-Tasten«.

Adresse 653 ($28D)

Tastencode der SHIFT-, CTRL- und Commodore-Taste

In der Speicherzelle 203 stehen die Codes aller Tasten, die gedrückt werden, außer die der drei Steuertasten SHIFT, CTRL und Commodore (oft auch CBM-, Logo- oder C=-Taste genannt). Diese drei Ausnahmen haben ihr eigenes Code-Register, eben 653.

Der Grund dafür liegt in der Bedeutung der drei Tasten. Sie können ja bekanntlich verschiedene Zeichensätze einschalten:

- SHIFT schaltet das Zeichen vorne rechts auf einer Taste ein
- C= schaltet das Zeichen vorne links auf einer Taste ein
- CTRL schaltet die Farben vorn auf den Zahlentasten ein
- SHIFT + C= schaltet von dem normalen Zeichensatz auf die Groß-/Kleinschreibung um.

Ich habe diese Zusammenhänge auch bei der Behandlung der Speicherzellen 245 und 246 erwähnt.

Die Codezahlen selbst sind auch in der Tabelle 9 enthalten. Der Vollständigkeit halber sind sie hier noch einmal angegeben:

SHIFT	1
C=	2
CTRL	4
SHIFT und C=	3
SHIFT und CTRL	5
C= und CTRL	6
SHIFT und C= und CTRL	7

Mit dem folgenden kleinen Programm und mit ein wenig Fingerfertigkeit können Sie diese Codewerte nachvollziehen:

```
10 PRINT PEEK(653)
20 GOTO 10
```

Eine interessante Anwendung habe ich im Texteinschub Nr. 21 »Abfrage der Tastencodes oder 476 Funktionstasten« gegeben.

Adresse 654 ($28E)

Tastencode der zuletzt gedrückten SHIFT-, CTRL- oder C=-Taste

Diese Speicherzelle wird zusammen mit der Zelle 653 verwendet, um zu verhindern, daß ein schlechter Tastendruck als mehrfaches Drücken derselben Taste gedeutet wird. Im Fachdeutsch nennt man das »Entprellen« einer Taste oder eines Kontaktes. Die Funktion ist vergleichbar mit der der Zelle 197 gegenüber der Zelle 203 für alle anderen Tasten.

Adresse 655 und 656 ($28F und $290)

Vektor auf die Routine der Tastencode-Tabellen

Das Betriebssystem hat eine Routine ab Adresse 60232 (60380 beim VC 20), auf die der Vektor in 655 und 656 zeigt. Sie liest den Codewert der SHIFT-, CTRL- und C=-Taste in der Speicherzelle 653 aus und verändert entsprechend den Vektor der Zellen 245 und 246, so daß er auf die richtige Codetabelle zeigt.

Texteinschub Nr. 26
Bunte Zeichen und bunter Hintergrund

1) Bunte Zeichen

Wie Zeichen und Buchstaben in bunten Farben auf den Bildschirm gedruckt werden, lernt jeder Hobby-Programmierer schon bei den ersten Gehversuchen - dasselbe innerhalb eines Programms zu erreichen, dauert sicher schon etwas länger.

Bei der Diskussion der Speicherzelle 646 habe ich drei Methoden dafür erwähnt. Ich habe auch gesagt, daß ich die Methode, den Farbcodewert in die Speicherzelle 646 zu POKEn, für die eleganteste halte. Deswegen verwendet das folgende Demonstrations-Programm dieses Verfahren, um den Bildschirm mit einer bunten Reihe der Zahl 1 zu füllen.

```
10 PRINT CHR$(147)
20 POKE 53281,1
30 FOR J=0 TO 1000
40 POKE 646,INT(RND(1)*14+2)
50 PRINT "1";
60 NEXT J
```

VC 20-Besitzer müssen die Zeilen 20, 30 und 40 umändern in:

```
20 POKE 36879,233
30 FOR J=0 TO 505
40 POKE 646,INT(RND(1)*6+2)
```

Erklärung:

Zeile 10 löscht den Bildschirm, Zeile 20 erzeugt einen weißen Hintergrund und eine hellblaue Umrahmung. Zeile 30 zählt vom ersten bis zum letzten Platz auf dem Bildschirm. Zeile 40 erzeugt für jedes Zeichen auf dem Bildschirm eine neue Farbe. Zeile 50 schließlich druckt, durch das Semikolon gesteuert, die Zahl 1 hintereinander, und zwar in den Farben, die in Zeile 40 zufällig ausgewürfelt wurden.

RND(1)*14 erzeugt eine Zufallszahl zwischen 0,1 und 13,99. Der Befehl INT davor macht daraus eine ganze Zahl zwischen 0 und 13. Um aber die Codezahl 1 für Weiß zu vermeiden, addieren wir noch 2 dazu, so daß wir Farbcodes zwischen 2 und 15 erhalten. Beim VC 20 ist das alles auf die Farben 2 bis 7 beschränkt.

Das Ergebnis ist wie gesagt ein Bildschirm voller Einser, deren Farben bunt wie ein Regenbogen abwechseln.

2) Bunter Hintergrund

Bunte Zeichen stellen also kein Problem dar. Wie steht es aber mit einem bunten Hintergrund? Den können wir zwar auch verändern (POKEn der Speicherzelle 53281 beziehungsweise 36879 beim VC 20), aber es bleibt immer nur »eintönig«.

Vom Commodore-Autor Jim Butterfield kenne ich nun eine Methode, die auch einen vielfarbigen Hintergrund bietet.

Butterfield geht dabei von einer lustigen Überlegung aus. Wir wissen zum Beispiel, daß der nächtliche Sternenhimmel aus hellen Punkten besteht, die vor einem schwarzen Hintergrund leuchten. Ohne dieses Wissen könnten wir aber ebenso gut annehmen, daß der Himmel - also der Hintergrund - im hellsten Weiß erstrahlt, aber durch einen schwarzen Vorhang (Vordergrund) mit vielen kleinen Löchern abgedunkelt ist.

Das folgende Demo-Programm benutzt diese Denkweise.

```
100 PRINT CHR$(147)
110 POKE 53281,1
120 FOR J=0 TO 1000
130 POKE 1024+J,160
140 POKE 55296+J,INT(RND(1)*14+2)
150 NEXT J
160 FOR K=0 TO 1000
170 POKE 1024+K,177
180 NEXT K
```

Für den VC 20 (ohne Erweiterung) sieht das Programm so aus:

```
100 PRINT CHR$(147)
110 POKE 36879,233
120 FOR J=0 to 505
130 POKE 7680+J,160
140 POKE 38400+J,INT(RND(1)*6+2)
150 NEXT J
160 FOR K=0 TO 505
```

```
170 POKE 7680+K,177
180 NEXT K
```

Die ersten drei Zeilen sind mit denen des ersten Demonstrations-Programms identisch.

Zeile 130 und 140 setzen auf jeden Platz des Bildschirms zuerst ein invertiertes Leerzeichen (Bildschirmcode 160), und zwar in einer der vielen möglichen Farben, per Zufallsgenerator in Zeile 140 ausgewählt.

Leerzeichen mit Farbe? Zugegeben, ein Leerzeichen hat normalerweise keine Farbe, man sieht es nicht. Das invertierte Leerzeichen hat aber eine Farbe. Sie kennen es vom Cursor, dessen Blinken dadurch erzeugt wird, daß das Leerzeichen zwischen normal und invertiert umgeschaltet wird (siehe auch die Beschreibung der Speicherzelle 647). Auf diese Weise besteht jetzt der Bildschirm aus einer Vielzahl von bunten Quadraten. Das ist der Vorhang von Jim Butterfield, der vor dem hellen weißen Hintergrund hängt.

Ab Zeile 160 werden alle Plätze des Bildschirms mit der invertierten 1 (Bildschirmcode 177) gefüllt. Diese invertierten Zeichen sind in der Farbe des Hintergrundes geschrieben, eben weiß. Dadurch entsteht der Eindruck, als wäre der Hintergrund bunt und die Zeichenfarbe weiß.

Der Eindruck verstärkt sich noch, wenn wir die 1 über den Bildschirm wandern lassen. Das erreichen wir durch Ändern der folgenden Zeilen:

```
170 POKE 1024+K,160
175 POKE 1025+K,177
```

Durch geschicktes Ausbauen der Zeile 140 können Sie einen vielfarbigen Bildschirm-Hintergrund in Zeilen oder Blöcken erzielen, ein weites Gebiet für bunte Grafik.

Texteinschub Nr. 27
Turbo-Tasten

Das Trio der Speicherzellen 650, 651 und 652 ist zuständig für die Steuerung der sogenannten Wiederholfunktion der Tasten. Darunter verstehen wir die Eigenschaft der Tastatur, das Zeichen oder die Funktion einer Taste so lange zu wiederholen, bis die Taste losgelassen wird. Normalerweise haben diese Funktion nur die Leertaste, die Cursor-Tasten und die INST/DEL-Taste.

Die Zahl in Speicherzelle 650 entscheidet, welche Tasten wiederholbar sind.

Schalten Sie bitte mit POKE 650,128 alle Tasten auf »wiederholbar« um.

Wenn Sie jetzt eine Taste drücken und sie festhalten, werden Sie folgendes beobachten können.

Nachdem das erste Zeichen auf dem Bildschirm erschienen ist, vergeht eine kurze Zeit, erst dann wird es mit einer gleichbleibenden Geschwindigkeit immer wieder ausgedruckt.

Für die anfängliche Verzögerung ist die Speicherzelle 652, für die Geschwindigkeit der nachfolgenden Wiederholungen die Speicherzelle 651 zuständig.

Viele Spieler und Anwender haben sich sicher schon oft gewünscht, sowohl die Reaktionszeit als auch die Geschwindigkeit der Wiederholfunktion beschleunigen zu können. Leider geht es in Basic nicht, weil die Zahlen in den Zellen 651 und 652 60mal in der Sekunde auf ihren ursprünglichen Wert zurückgesetzt werden.

Aber in Maschinensprache geht es sehr wohl, und zwar mit der sogenannten Interrupt-Methode. Über sie und ihre Wirkungsweise ist schon ausführlich berichtet worden: von Boris Schneider in Ausgabe 3/85 (Der gläserne VC 20) und von Heimo Ponnath in den Ausgaben 7/85 und 8/85 (Assemblerkurs). Ich werde hier nur innerhalb der Beschreibung des folgenden Kochrezeptes darauf eingehen.

Das Kochrezept zur Veränderung der Inhalte von 651 und 652 stammt von Dan Carmichael aus seinem Aufsatz »Speeding Up The VIC« in Ausgabe 10/83 der COMPUTE!'s Gazette.

Wir schreiben es als Maschinenprogramm in Form von DATA-Zeilen in den Bandpuffer ab Adresse 828, wo es geschützt residieren kann, solange keine Kassettenoperationen durchgeführt werden. Das Ladeprogramm in Basic steht in Listing 1 (Seite 63). Für den VC 20 lautet die vorletzte Zahl 191 statt 49.

In Listing 2 (Seite 63) ist das Programm disassembliert dargestellt.

Beim VC 20 lautet der Sprungbefehl in Zelle 851 JMP 60095.

Für Anhänger der hexadezimalen Darstellung gebe ich das Programm als HEX-Ausdruck in Listing 3 (Seite 63) wieder.

Für den VC 20 lautet die letzte Zeile anders:

,0353 4C BF EA JMP EABF

Mit dem Befehl SEI werden jegliche Programmunterbrechungen gesperrt. Anschließend kommt das Zahlenpaar 73 und 3 in die Speicherzellen 788 und 789, wo es in Low/High-Byte Darstellung die Adresse 841 (73+256*3=841) darstellt.

In 788 und 789 steht normalerweise ein Vektor auf die Adresse 59953 (60095 beim VC 20), von der aus die Aufgaben der »normalen« Unterbrechungsroutine gesteuert werden. Wir »verbiegen« also den Vektor so, daß er auf die Speicherzelle 841 zeigt.

Die schon genannte Unterbrechungsroutine, die 60mal pro Sekunde alles unterbricht, um die STOP-Taste abzufragen, die Uhr weiterzuschalten und so weiter, springt jetzt nicht auf 59953, sondern zuerst nach 841.

Ab 841 steht jedoch der zweite Teil unseres Maschinenprogramms, das die eingangs gewünschte 1 beziehungsweise 0 nach 651 und 652 schreibt. Das erfolgt jetzt laufend, ein Effekt, der uns in Basic verwehrt ist. Danach allerdings kommt ein letzter Sprungbefehl, der dort weitermacht, wo die Unterbrechunsroutine ursprünglich hätte fortfahren sollen, nämlich in 59953 (60095).

Jetzt fehlt nur noch die Beschreibung, wie sich das alles auswirkt. Ich nehme an, Sie haben immer noch mit POKE 650,128 die gesamte Tastatur auf Wiederholfunktion geschaltet, wenn nicht, holen Sie es bitte nach. Laden Sie das Basic-Programm von Listing 1 und starten Sie es mit RUN. Jetzt steht es in den Speicherzellen 828 bis 853 und kann mit SYS 828 gestartet werden.

Wenn Sie jetzt wieder eine Taste länger gedrückt halten, flitzt das entsprechende Zeichen wie ein Turbo-Auto über den Bildschirm. Der Cursor ist mit den Augen fast nicht mehr zu verfolgen. Es geht alles so schnell, daß Sie Mühe haben, nur ein einzelnes Zeichen auf den Bildschirm zu bringen. Wenn Sie das wollen: Mit RUN/STOP und RESTORE stellen Sie den ursprünglichen Zustand wieder her.

Das kleine Maschinenprogramm läßt sich in jedes Spiel oder Anwendungsprogramm nutzbringend einbauen.

Es gibt Anwenderprogramme, die diesen Vektor so verbiegen, daß die Decodierung der Tasten umgangen und durch eine andere, selbstgebaute Routine ersetzt wird. So kann zum Beispiel das Drücken einer bestimmten Taste umgemünzt werden in Ausdrucken von Basic-Befehlen auf dem Bildschirm.

Adresse 657 ($291)

Flagge für Verriegelung der Zeichensatz-Umschaltung

Durch gleichzeitiges Drücken der SHIFT- und der Commodore-Taste wird bekanntlich der Zeichensatz 1 (Großbuchstaben und Grafik-Zeichen) umgeschaltet auf den Zeichensatz 2 (Groß- und Kleinbuchstaben), ein zweites Drücken der beiden Tasten schaltet den Zeichensatz zurück.

Diese Umschaltung wird verriegelt, wenn in der Speicherzelle 657 eine 128 steht. Eine 0 läßt die Umschaltung zu.

Dieser Effekt kann auf zwei, beim C 64 sogar auf drei Arten erzielt werden:

☐ Umschaltung des Zeichensatzes zulassen

```
POKE 657,0
PRINT CHR$(9)
CTRL und I (nur C 64)
```

☐ Umschaltung des Zeichensatzes verriegeln

```
POKE 657,128
PRINT CHR$(8)
CTRL und H (nur C 64)
```

Adresse 658 ($292)

Flagge für Scrollen

Die Flagge in dieser Speicherzelle legt fest, ob eine weitere echte Zeile zu einer logischen Zeile hinzugefügt wird, sobald der Cursor über das 40ste Zeichen der Zeile (22ste Zeichen beim VC 20) hinausläuft.

Steht in 658 eine 0, dann werden alle Zeilen hochgeschoben (man nennt das »scrollen«), um der neuen Zeile Platz zu machen.

Wenn in der Zeile irgendein Wert größer als Null steht, unterbleibt dieses Scrollen. Die Flagge wird immer dann auf den höheren Wert gesetzt, wenn Zeichen im Tastaturpuffer (631 bis 640) stehen und darauf warten, am Ende des Programms ausgedruckt beziehungsweise ausgeführt zu werden. Diese Verriegelung wird deshalb eingesetzt, weil im Tastaturpuffer Zeichen wie zum Beispiel Cursor-Bewegungen stehen können.

Von Basic aus kann diese Speicherzelle nicht beeinflußt werden.

Die RS232-Schnittstelle

Wir haben schon früher Speicherzellen besprochen, die mit der RS232-Schnittstelle zu tun haben, ohne die letztere dabei genauer zu betrachten. Da der heutige Teil des Kurses jedoch die zehn wichtigsten Speicherzellen behandelt, welche diese Schnittstelle betreffen, komme ich nicht umhin, näher auf sie einzugehen. Ich tue das mit voller Absicht, obwohl dadurch die erklärenden Texteinschübe mehr Platz einnehmen als die Besprechung der Speicherstellen selbst. Falls Sie mit Schnittstellen des Computers nicht vertraut sind, sollten Sie zuerst den Texteinschub Nr. 28 »Schnittstelle und Port« anschauen, mit dem ich die gängige Begriffsverwirrung klären möchte.

Adresse 659 ($293)

RS232-Steuerregister

Jeder OPEN-Befehl, mit dem bekanntlich eine Datei (File) eröffnet wird, kann neben File-Nummer und Geräte-Nummer auch einen File-Namen haben. Der File-Namen einer RS232-Schnittstelle hat maximal nur vier Zeichen. Das erste Zeichen wird in diese Speicherzelle 659 gebracht und steuert dort Übertragungsgeschwindigkeit, die Wortlänge und die Anzahl der Stopp-Bits. Die nähere Bedeutung dieser Fachwörter können Sie dem Texteinschub Nr. 29 »Die Elemente der RS232-Schnittstelle« entnehmen. Tabelle 11 zeigt die Bedeutung jedes einzelnen Bits dieser Speicherzelle.

Die praktische Anwendung dieser Bit-Werte innerhalb eines OPEN-Befehls ist ausführlich im Texteinschub Nr. 30 »Die Programmierung der RS232-Schnittstelle« beschrieben.

Adresse 660 ($294)

RS232-Befehlsregister

In diese Speicherzelle wird, ähnlich wie bei der Zelle 659, das zweite Zeichen des File-Namens gebracht. Die einzelnen Bits steuern das Handshake-Protokoll (3-/X-Leitung), den Duplex-Modus (Halb-/Voll-Duplex) und die Parity-Prüfung (keine, gerade, ungerade). Die nähere Bedeutung dieser Fachwörter können Sie dem Texteinschub Nr. 11 »Die Elemente der RS232-Schnittstelle« entnehmen. Tabelle 12 zeigt die Bedeutung jedes einzelnen Bits dieser Speicherzelle.

Wenn Sie sich für die praktische Anwendung dieser Bit-Werte innerhalb eines OPEN-Befehls interessieren, dann verweise ich an dieser Stelle auf den Texteinschub Nr. 30 »Die Programmierung der RS232-Schnittstelle«.

Adresse 661 und 662 ($295 und $296)

RS232 frei wählbare Übertragungsgeschwindigkeit

Es war ursprünglich vorgesehen, durch entsprechende Wahl des dritten und vierten Zeichens im File-Namen beliebige Übertragungsgeschwindigkeiten einzustellen. Die jeweiligen Werte sollten die Speicherzellen 661 und 662 enthalten. Diese Möglichkeit wurde aber nicht eingebaut. Der Grund dafür dürfte wohl der sein, daß die wählbaren Übertragungsgeschwindigkeiten aller Geräte auf bestimmte Werte normiert sind.

Adresse 663 ($297)

RS232-Statusregister

Genauso wie in der Speicherzelle 144 der Status aller Ein- und Ausgabe-Operationen angezeigt wird, werden alle Fehler der RS232-Schnittstelle in der Speicherzelle 663 angezeigt. Die Bedeutung der einzelnen Bits, wenn sie auf 1 gesetzt sind, zeigt Tabelle 14.

Der Status wird nicht automatisch angezeigt, sondern muß vom Programm abgefragt werden. Abfragen können Sie sowohl durch PEEKen der Speicherzelle 663 als auch durch Aufrufen der Statusvariablen ST. Die Variable ST, die normalerweise den Inhalt der Zelle 144 wiedergibt, schaltet nach dem Eröffnen eines RS232-Kanals durch OPEN 1,2 auf die Speicherzelle 663 um. Jedoch ist Vorsicht geboten, da durch Aufruf von ST der Inhalt von 663 gelöscht wird.

Es ist ratsam, den Wert von ST erst einer anderen Variablen zuzuordnen, wenn sie mehrfach verwendet werden soll.

Falls das Status-Register einen Fehler anzeigt, muß das Programm entsprechende Konsequenzen ziehen. Wenn zum Beispiel Bit 0 oder Bit 1 gesetzt sind, ist es angebracht, das letzte Daten-Byte noch einmal zu übertragen. Wenn Bit 2 gesetzt ist, heißt dies, daß der GET#-Befehl den Eingabepufferspeicher nicht schnell genug entleert. Falls die Übertragungsgeschwindigkeit von 300 Bit/s, die maximal mit einem Basic-Programm erreichbar ist, nicht ausreicht, muß entweder der Sender langsamer eingestellt werden, oder Sie schrei-

BINÄR	BITWERT	STEUERFUNKTION
Bit 0 bis 3 steuern die Übertragungsgeschwindigkeit		
xxxx0000	0	(siehe 1)
xxxx0001	1	50 Bit/s
xxxx0010	2	75 Bit/s
xxxx0011	3	110 Bit/s
xxxx0100	4	134,5 Bit/s
xxxx0101	5	150 Bit/s
xxxx0110	6	300 Bit/s
xxxx0111	7	600 Bit/s
xxxx1000	8	1200 Bit/s
xxxx1001	9	1800 Bit/s
xxxx1010	10	2400 Bit/s
die Werte 11 bis 15 sind nicht belegt		
Bit 4 ist nicht belegt		
Bit 5 und 6 steuern die Wortlänge		
x00xxxxx	0	8-Bit-Wort
x01xxxxx	32	7-Bit-Wort
x10xxxxx	64	6-Bit-Wort
x11xxxxx	96	5-Bit-Wort
Bit 7 steuert die Stopp-Bits		
0xxxxxxx	0	1 Stopp-Bit
1xxxxxxx	128	2 Stopp-Bit

Fußnote (1)
Bei diesem Bitwert sollte ursprünglich der Anwender in Speicherzelle 661 eigene Bandraten wählen können. Diese Möglichkeit ist aber nicht realisiert worden.

Tabelle 11. Die Bedeutung der einzelnen Bits im RS232-Steuerregister

BINÄR	BITWERT	STEUERFUNKTION
Bit 0 steuert den Handshake-Typ		
xxxxxxx0	0	3-Leitung
xxxxxxx1	1	X-Leitung
Bit 1 bis 3 sind nicht belegt		
Bit 4 steuert den Duplex-Modus		
xxx0xxxx	0	Voll-Duplex
xxx1xxxx	16	Halb-Duplex
Bit 5 bis 7 steuern den Paritäts-Test		
000xxxxx	0	keine Parität
001xxxxx	32	ungerade (1)
010xxxxx	64	keine Parität
011xxxxx	96	gerade (1)
100xxxxx	128	keine Parität
101xxxxx	160	Parität=1 (2)
110xxxxx	192	keine Parität
111xxxxx	224	Parität=0 (2)

Fußnoten:
(1) vom Sender berechnet, vom Empfänger geprüft
(2) vom Sender festgelegt, vom Empfänger nicht geprüft

Tabelle 12. Das RS232-Befehlsregister im einzelnen

ben das Programm in Maschinensprache.

Adresse 664 ($298)

RS232 - Anzahl der zu übertragenden Bits

Diese Speicherzelle wird verwendet, um festzustellen, mit wievielen Nullen das zu übertragende Zeichen aufgefüllt werden muß, um die in Speicherzelle 659 (Bit 5 und 6) ausgewählte Wortlänge herzustellen (s. Speicherzellen 168 und 180).

BIT	BIT-WERT	BEDEUTUNG
0	1	Fehler bei Paritäts-Test
1	2	Fehler in der Bit-Folge
2	4	Überlauf des Eingabepufferspeichers
3	8	Eingabepufferspeicher ist leer
4	16	das Clear-To-Send (CTS)-Signal (Handshake) fehlt
5	32	nicht belegt
6	64	das Data-Set-Ready (DSR)-Signal (Handshake) fehlt
7	128	Übertragung ist unterbrochen

Tabelle 14. Das RS232-Statusregister

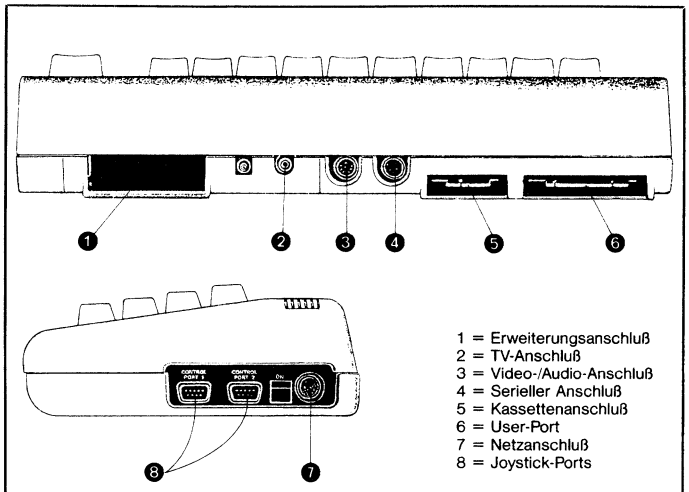


Bild 23. Anschlüsse des C 64

Texteinschub Nr. 28
Schnittstelle und Port

Immer wenn die Rede davon ist, den Computer mit irgendwelchen Geräten zu verbinden, tauchen Fachwörter auf, wie Interface, Schnittstelle, Port, Eingang, Ausgang und Stecker. Da im Kurs gerade die Speicherzellen behandelt werden, die für die RS232-Schnittstelle zuständig sind, möchte ich die Gelegenheit nutzen, ein wenig Klarheit in dieses Begriffswirrwarr zu bringen. Zuerst sollen die Begriffe erklärt werden:

1) »Interface« ist das englische Wort für »Schnittstelle«.

In einer Schnittstelle sind die Regeln und Vorschriften festgelegt, wie Daten zwischen zwei Geräten (zum Beispiel Computer und Floppy) ausgetauscht werden. Festgelegt ist hauptsächlich:
- ob ein Datenwort auf einen Schlag (parallel) oder jedes Bit einzeln (seriell) übertragen wird
- die Geschwindigkeit der Übertragung
- die Signale, mit denen die beteiligten Geräte den Ablauf der Übertragung steuern
- mit welchen Spannungs- oder Stromwerten die binäre 1 beziehungsweise 0 dargestellt wird
- die elektrischen Spannungen und Ströme, die bei der Übertragung maximal auftreten dürfen

Sie sehen, eine Schnittstelle ist in erster Linie eine Anzahl von Regeln. Manchmal allerdings werden auch die Module und Spezialkabel, welche die Regeln technisch in die Tat umsetzen, Schnittstellen genannt.

2) Über einen »Ausgang« kann der Computer (oder ein anderes Gerät) Daten abgeben, über einen »Eingang« erhält er Daten. Ein »Port« ist beides, Ein- und Ausgang. Ein »Stecker« schließlich ist die technische Ausführung der Verbindung.

So, nach dieser Begriffserklärung wollen wir uns anschauen, welche Ports, Ein- und Ausgänge, der Computer hat. Die Zeichnung dieser Anschlüsse (Bild 23) habe ich dem Commodore-Handbuch entnommen, nicht aber ihre Bezeichnungen, denn diese gehen bereits wild durcheinander.

Der »Erweiterungsanschluß« (1) wird hauptsächlich als Eingang für Spielmodule verwendet. Er ist aber ein echter Port, nicht zuletzt zur Speichererweiterung beim VC 20 und kann für Schnittstellen über entsprechende Routinen des Betriebssystems programmiert werden.

Der »TV-Anschluß« (2) ist ein reiner Ausgang des im Computer eingebauten Fernsehmodulators, der beim VC 20 fehlt.

Der »Video/Audio-Anschluß« (3) ist ebenfalls ein reiner Ausgang der Ton- und Bildsignale für einen Monitor oder für den externen Fernsehmodulator des VC 20.

Der »Serielle Anschluß« (4) ist ein Port, über den das Diskettenlaufwerk und Drucker angeschlossen werden. Er ist für Schnittstellen programmierbar.

Der »Kassettenanschluß« (5) ist ebenfalls ein Port, der speziell für die Datasette eingerichtet ist. Bastler und Tüftler, die Schaltpläne lesen können und das Betriebssystem des Computers kennen, müßten in der Lage sein, mit diesem Port auch andere externe Geräte zu steuern. Für die genormten Schnittstellen kommt er meines Wissens nicht in Frage.

Der »User-Port« (6) ist das, was sein Name sagt, nämlich ein Port für verschiedene Anwendungen und Schnittstellen. Er ist über 16 Register des 6526 Complex Interface-Adapters (CIA) mit den Adressen 56320 bis 563215 frei programmierbar. Der VC 20 hat dafür einen 6522 Versatile Interface-Adapter (VIA), dessen 16 Register die Adressen 37136 bis 37151 haben.

Der »Netzanschluß« (7) ist ein reiner Eingang, aber nicht für Daten.

Die »Spielanschlüsse« (8) (nur einer beim VC 20) werden eigentlich nur als Eingang für Joysticks, Lichtgriffel und Paddles (Drehregler) verwendet, obwohl sie vom Prinzip her programmierbare Ports sind. Ihre universelle Verwendung ist sicher nur Spezialisten vorbehalten.

Zuletzt sollen auch die verbreitetsten Schnittstellen noch erwähnt werden. International haben sich besonders die folgenden drei Schnittstellen durchgesetzt:
- die RS232-Schnittstelle
- die IEC/IEEE-488-Schnittstelle
- die Centronics-Schnittstelle

Die »RS232-Schnittstelle« ist eine serielle Schnittstelle. Sie wurde schon vor 100 Jahren als »TTY-Version« zur Textübertragung mit Fernschreibern eingerichtet, bei der die logische 0 durch einen Strom von 20 Milliampere und die 1 durch keinen Strom dargestellt wurde. Heute wird fast nur noch die »V.24-Version« zur Datenfernübertragung verwendet, bei der die 0 durch eine positive Spannung zwischen 3 und 15 Volt, die 1 aber durch eine entsprechende negative Spannung dargestellt wird.

Beim C 64 und VC 20 ist die RS232-/V.24-Schnittstelle am User-Port verfügbar, allerdings nicht mit den oben genannten Spannungswerten für die 0 und 1. Dieses für Commodore typische Sparverfahren macht eine zusätzliche Signalumsetzung erforderlich. Die RS232-Schnittstelle wird hauptsächlich für Datenübertragung per Modem oder Akustikkoppler eingesetzt. Ihr Arbeitsprinzip ist im Texteinschub Nr. 29 »Die Elemente der RS232-Schnittstelle« beschrieben.

Zur parallelen Datenübertragung entstand in Europa die »IEC-625-Schnittstelle«, in USA die »IEEE-488-Schnittstelle«. Beide sind praktisch identisch. Sie unterscheiden sich nur in der Verwendung des Steckers (was natürlich idiotisch ist). Bei den Commodore-Computern ist diese Schnittstelle sowohl am Erweiterungs-Port (1) als auch über den seriellen Port (4) einrichtbar. Der serielle Port allerdings enthält wiederum eine für Commodore typische Einschränkung. Er erlaubt, wie sein Name

andeutet, nur eine serielle Datenübertragung. Das heißt, statt – wie bei der IEC-/IEEE-Schnittstelle festgelegt – alle 8 Bit eines Wortes über acht Leitungen gleichzeitig, werden hier die Bits hintereinander auf nur einer Leitung übertragen. Auch das erfordert eine zusätzliche Anpassung. Das Prinzip der IEC-/IEEE-Schnittstelle wurde bereits in Ausgabe 3/85 ab Seite 24 von Arnd Wängler genau beschrieben.

Die »Centronics-Schnittstelle« ist aus der harten Realität des Geschäftslebens entstanden. Während sich noch die Normstellen in Europa und USA herumstritten, hat der damalige Marktführer unter den Druckerherstellern, die Firma Centronics, eine eigene Schnittstelle geschaffen, die sich schlicht und einfach durch die weite Verbreitung der Centronics-Drucker durchgesetzt hat. Sie ist eine parallele Schnittstelle, die sich beim C 64/VC 20 sowohl am User-Port (6) als auch am seriellen Port (4) einrichten läßt. Zur Beschreibung der Schnittstelle habe ich in meiner Literatursammlung nur zwei Aufsätze gefunden, einen von Georg Werner in c't, Ausgabe 4/84, Seite 92, einen anderen von Peter Bonsch in Computer Persönlich, Ausgabe 11/83 ab Seite 152. Tabelle 13 gibt eine Zusammenfassung über die Realisierbarkeit der drei Schnittstellen an den Ports von C 64 und VC 20.

SCHNITT-STELLE	PORT	Erweit. (1)	Seriell (4)	Kassett. (5)	User (6)	Spiel (8)
RS232					X	
IEC/IEEE		X	X			
Centronics			X		X	

Tabelle 13. Realisierbarkeit der Schnittstellen am C 64 und VC 20

Texteinschub Nr. 29
Die Elemente der RS232-Schnittstelle

Da meine Texteinschübe kurz sein sollen, beschränke ich mich auf Erklärungen der Vorgänge, die mit den im Kurs behandelten Speicherzellen 659 bis 670 zu tun haben. Weitere Erläuterungen können Sie dem Aufsatz von Jens Maßmann der Ausgabe 5/85, Seite 80 entnehmen.

Wortlänge: Die Schnittstelle ermöglicht die Übertragung von Datenwörtern (Bytes), deren Länge vor der Übertragung eingestellt werden kann. Es sind Wortlängen von 5, 6, 7 oder 8 Bit erlaubt. Die Wortlänge wird durch Bit 5 und 6 der Speicherzelle 659 eingestellt.

Übertragungsgeschwindigkeit (Baud-Rate): Daten werden seriell übertragen, das heißt alle Bits eines Datenwortes (Byte) laufen hintereinander über eine Leitung zum Empfänger. Dabei ist wesentlich, daß Sender und Empfänger sich einig sind, mit welcher Geschwindigkeit die Bit-Kette übertragen wird. Diese Übertragungsgeschwindigkeit, auch Baud-Rate genannt, wird in Bit/s angegeben. Die Übertragungsgeschwindigkeit wird durch Bit 0 bis 3 der Speicherzelle 659 eingestellt und reicht von 110 bis 2400 Bit/s.

Stopp-Bits: Die Übertragungsgeschwindigkeit muß sowohl im Sender als auch im Empfänger der Daten fest eingestellt werden. Da die beiden Geräte dies unabhängig voneinander tun, besteht die Gefahr, daß diese Einstellungen nicht ganz genau gleich sind. Das könnte zur Folge haben, daß sie nach vielleicht 2000 Bit um 1 Bit auseinanderliegen. Alle nachfolgenden Übertragungen wären dann völlig falsch.

Deshalb wird die Übertragung nach jedem Wort neu eingestellt, man nennt das »synchronisieren«. Dazu dient am Anfang eines Wortes ein Start-Bit und am Ende eines Wortes ein oder zwei Stopp-Bits. Die Stopp-Bits definieren den Ruhezustand (logische 1) der Übertragungsleitung, ihre Anzahl die minimale Zeit des Ruhezustandes. Sobald ein neues Wort mit einem Start-Bit (logische 0) beginnt, übernimmt der Empfänger den Übergang von 0 nach 1 als Startimpuls für die Abfrage der nächsten ankommenden Bits. Die Anzahl der Stopp-Bits werden durch Bit 7 der Speicherzelle 659 eingestellt.

Paritätsprüfung: Zusätzlich zu den Datenbits und den Start-/Stopp-Bits können sogenannte Paritätsbits übertragen werden. Sie ermöglichen eine grobe Fehlerkontrolle. Der Sender errechnet die Quersumme aller Datenbits. Bei der sogenannten »geraden« Paritätsprüfung wird das Paritätsbit so gewählt, daß es die Quersumme zu einer geraden Zahl ergänzt, bei der »ungeraden« Paritätsprüfung ist es gerade umgekehrt. Der Empfänger macht dieselbe Rechnung und vergleicht sein Paritätsresultat mit dem empfangenen Paritätsbit des Senders. Sie sollten natürlich gleich sein. Auf diese Weise können einfache Übertragungsfehler erkannt werden. In den Commodore-Computern sind noch zwei weitere Möglichkeiten eingebaut, nämlich das Paritätsbit ohne Quersummenrechnung immer auf 1 oder aber immer auf 0 zu setzen. Mit den Bits 5 bis 7 der Speicherzelle 660 können insgesamt vier verschiedene Paritätsprüfungen eingestellt werden (siehe auch Texteinschub Nr. 18).

Duplex-Modus: Sind zwei Geräte, von denen eines nur empfangen, nicht aber selbst senden kann, über eine Leitung verbunden, nennt man diese Einbahnstraße eine Simplex-Verbindung. Können aber beide Geräte senden und empfangen, spricht man von einer Duplex-Verbindung.

Duplex gibt es in zwei Betriebsarten. Der Voll-Duplex-Modus erlaubt ein gleichzeitiges Senden beider Geräte. Im Halb-Duplex-Modus kann immer nur ein Gerät senden, allerdings wechselweise. Der Duplex-Modus wird durch Bit 4 der Speicherzelle 660 eingestellt.

Handshake-Protokoll: Mit Handshake (Händeschütteln) wird ein Verfahren bezeichnet, bei dem zwei Geräte sich gegenseitig durch gesonderte Signale mitteilen, ob sie bereit sind, Daten abzusenden beziehungsweise zu empfangen. Die Festlegung der zeitlichen Reihenfolge dieser Handshake-Signale nennt man Protokoll.

Dieses Verfahren hat den Vorteil, daß Sender und Empfänger völlig unabhängig voneinander ihr eigenes Programm ausführen können und nur selten aufeinander warten müssen. Voraussetzung ist allerdings ein Pufferspeicher (siehe unten). Es gibt zwei Arten von Handshakes, den 3-Leitungs-Handshake (auch Rückkanal-Handshake genannt) und den X-Leitungs- oder Voll-Handshake.

Der 3-Leitungs-Handshake braucht, wie der Name sagt, nur drei Leitungen: für gemeinsame Erde (Masse), für die gesendeten Daten und für die empfangenen Daten. Der Handshake besteht darin, daß der Empfänger auf der freien Leitung, eben dem Rückkanal, dem Sender durch je ein Zeichen mitteilt, wenn er bereit ist, Daten zu übernehmen oder wenn er keine Daten übernehmen kann.

Der X-Leitungs-Handshake stellt viel mehr Leitungen zur Verfügung, nämlich die gleichen drei wie vorher, zusätzlich aber pro Sender und Empfänger mehrere Leitungen für Anmeldung und Rückmeldung der Bereitschaft, sowie für die Ausführung der Übertragung. Es gibt theoretisch insgesamt 25 Leitungen für die RS232-Schnittstelle, beim C 64 beziehungsweise VC 20 sind aber nur zehn ausgeführt. Das Handshake-Protokoll kann durch Bit 0 der Speicherzelle 660 ausgewählt werden.

Pufferspeicher: Immer wenn ein RS232-Kanal geöffnet wird, zwackt das Betriebssystem des Computers dem Programmspeicher am oberen Ende zwei Pufferspeicher ab, mit einer Größe von je 256 Byte für empfangene und zu sendende Daten. Diese First-In-First-Out-Speicher (die als erste eingespeicherten Daten werden auch als erste wieder ausgelesen) sind als dynamische Ringspeicher aufgebaut. Statt zu warten, bis der Empfänger zur Datenübernahme bereit ist, schreibt der Sender die Daten in den Pufferspeicher, aus dem die Schnittstelle sie an den Empfänger weitergibt, sobald dieser bereit ist.

Dieses fast ungeordnete Füllen und Leeren des Pufferspeichers hat zur Folge, daß Beginn und Ende des Speichers je nach Datenmenge innerhalb der 256 Byte stets in Bewegung sind. Um jederzeit die Anfangs- und Endadressen feststellen zu können, werden sie in den Speicherzellen 667 bis 670 mitgezählt.

Statusregister: In der Speicherzelle 663 werden alle Fehler einer RS232-Übertragung festgehalten. Jedes Bit hat eine eigene Bedeutung, die in der Tabelle bei der Beschreibung der Speicherzelle 663 angegeben ist. Diese Fehler werden leider nicht, wie im Basic, automatisch angezeigt. Sie müssen vielmehr ausgelesen und identifiziert werden, um dann im Programm mit entsprechenden Maßnahmen korrigiert zu werden.

Texteinschub Nr. 30
Die Programmierung der RS232-Schnittstelle

Die Programmierung der RS232-Schnittstelle ist denkbar einfach. Alle dazu notwendigen Routinen sind im Betriebssystem des Computers bereits enthalten. Das genau macht ja die Schnittstelle so attraktiv. Die Schnittstelle verwendet genau dieselben Befehle wie die serielle Schnittstelle, über die der Computer mit Floppy und Drucker verbunden ist, nämlich OPEN, CMD, PRINT#, GET#, INPUT# und CLOSE. Auch die Statusvariable ST wird herangezogen. Wichtig ist jedoch, daß die RS232-Schnittstelle die Gerätenummer 2 hat.

Eröffnung des RS232-Kanals

Wie gewohnt, wird er mit dem OPEN-Befehl geöffnet:

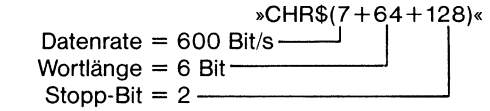
OPEN File-Nr, Geräte-Nr, Datenkanal-Nr, File-Name

- die File-Nummer kann Werte von 0 bis 255 annehmen, wie bei jedem OPEN-Befehl auch
- die Geräte-Nummer ist immer 2
- der Wert der Datenkanal-Nummer ist bedeutungslos, da immer nur ein RS232-Kanal offen sein darf. Wird zusätzlich ein zweiter Kanal geöffnet, werden die Daten des ersten Kanals im Pufferspeicher zerstört.
- der File-Name hat hier eine besondere und entscheidende Bedeutung. Er besteht aus maximal vier Zeichen. Der ASCII-Wert des ersten Zeichens wird in die Speicherzelle 659 übertragen und legt dadurch die Übertragungsgeschwindigkeit, die Wortlänge und die Anzahl der Stopp-Bits fest (siehe Texteinschub Nr.29). Der ASCII-Wert des zweiten Zeichens gelangt in die Speicherzelle 660 und bestimmt dadurch das Handshake-Protokoll, den Duplex-Modus und die Paritätsprüfung (siehe Texteinschub Nr.29). Zeichen 3 und 4 sind nicht festgelegt.

Man kann den File-Namen des OPEN-Befehls in zwei Arten schreiben, die natürlich identisch sind:

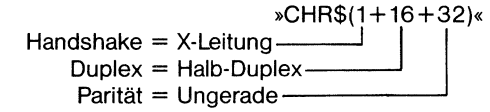
```
(1) OPEN 1,2,0,CHR$(7+64+128) + CHR$(1+16+32)
(2) OPEN 1,2,0,CHR$(199)+CHR$(49)
```

Theoretisch könnte man noch eine dritte Schreibweise hernehmen, nämlich die Zeichen hinschreiben, die den ASCII-Wert 199 beziehungsweise 49 haben. Dann käme die Schreibweise einem File-Namen noch am nächsten. Ein Blick in die Tabelle der ASCII-Codes belehrt uns aber eines Besseren, da wir Zweideutigkeiten nicht ausschließen können. Also ist die Schreibweise der Zeichen mit ihren ASCII-Werten doch am besten. Ich persönlich ziehe die Schreibweise (1) vor, da wir aus ihr sofort die dadurch definierten Werte ablesen können.



Sie können die Zusammenhänge direkt der Tabelle 11 entnehmen, die bei der Erklärung der Speicherzelle 659 steht.

Entsprechend wird aus der Tabelle 12 der Speicherzelle 660 das zweite Zeichen »CHR$(1+16+32)« zusammengesetzt:



Der OPEN-Befehl mit Gerätenummer 2 hat noch eine Besonderheit, die ich schon bei der Besprechung der Speicherzellen 55 und 56 erwähnt habe. Sobald er nämlich im Programm auftaucht, wird durch ihn der Zeiger in 55 und 56, der ja das obere Ende des Programmspeichers angibt, um 512 Byte nach unten geschoben, um Platz für die beiden Pufferspeicher zu schaffen.

Wenn das mitten im Programm passsiert und vorher schon Zeichenketten (Strings) definiert worden sind (die bekanntlich vom oberen Ende des Speichers aus angelegt werden), werden diese überschrieben.

Also Vorsicht: Wer beabsichtigt, in einem Programm eine RS232-Schnittstelle zu aktivieren, soll diese unbedingt am Anfang des Programms öffnen, damit der Speicherplatz richtig zugeordnet wird.

Daten an den RS232-Kanal übergeben

Die Daten werden zuerst in den Ausgabepuffer gebracht, von dort gelangen sie, vom Handshake-Protokoll gesteuert, an den Empfänger. Die Befehle dazu sind CMD und PRINT#.

»CMD File-Nr.,Zeichen« schaltet vom Bildschirm auf den RS232-Empfänger um.

»PRINT# File-Nr, Zeichen« schreibt die Zeichen in den Ausgabepufferspeicher, von wo sie die Schnittstelle automatisch herausholt.

Beide Befehle wirken genauso wie bei anderen Dateien. Vorsicht ist jedoch geboten, wenn laufend Daten in den Pufferspeicher geschrieben werden, ohne zu wissen, ob die Schnittstelle den Puffer auch wieder entleert hat. Bei Überlauf des Puffers gehen Daten verloren. Es ist ratsam, durch Vergleich der beiden Indizes in den Speicherzellen 669 und 670, die Anfang und Ende des Ausgabepufferspeichers markieren, auf Überlauf zu prüfen.

Daten vom RS232-Kanal übernehmen

Daten, die von der Schnittstelle in den Eingabepufferspeicher gebracht worden sind, werden mit INPUT# oder GET# ausgelesen:

INPUT# File-Nr, Zeichen

GET# File-Nr, Zeichen

Auch hier kann ein Überlaufen des Pufferspeichers auftreten, wenn nämlich die Schnittstelle mehr oder schneller Daten liefert, als mit GET# oder INPUT# ausgelesen werden können. Dieser Zustand kann sowohl durch Überprüfung der Indizes in den Speicherzellen 667 und 668 als auch durch Prüfung von Bit 2 des Statusregisters in 663 erkannt beziehungsweise vermieden werden. Der Speicher kann auch leer sein. Bei Verwendung von INPUT# wartet der Rechner und stürzt bei abgeschalteter Schnittstelle ab. Es ist deshalb empfehlenswert, immer den Befehl GET# zu verwenden, der bei leerem Speicher höchstens einen Nullstring (" ") liefert. Bit 3 des Statusregisters prüft diesen Fall.

Schließen des RS232-Kanals

Der Befehl »CLOSE File-Nr.« schließt den Kanal. Dabei werden die Ein- und Ausgabe-Pufferspeicher aufgelöst, indem der Zeiger in Speicherzelle 55 und 56 wieder auf das Ende des Programmspeichers zeigt. Alle Handshake-Leitungen werden in den Ruhezustand gesetzt und alle Datenübertragungen unterbunden.

Programm-Beispiel

Für eine echte Demonstration müßten Sie eine RS232-Schnittstelle über den User-Port eingerichtet haben. Da ich das nicht voraussetzen kann, begnüge ich mich damit, das im Programmierhandbuch von Commodore angegebene Beispiel zu bringen und zu kommentieren.

```
10 OPEN 1,2,0,CHR$(6+32)+CHR$(32+128)
20 GET#1,A$
30 GET B$
40 IF B$ <> "" THEN PRINT#1,B$;:PRINT B$
50 GET#1,C$
60 PRINT C$;
70 PR = PEEK(663)
80 IF PR=0 OR PR=8 THEN 30
100 IF PR AND 1 THEN PRINT "PARITY-FEHLER"
110 IF PR AND 2 THEN PRINT "BITFOLGE-FEHLER"
120 IF PR AND 4 THEN PRINT "EINGABESPEICHER VOLL"
130 IF PR AND 128 THEN PRINT "UNTERBRECHUNG"
```

Zeile 10 öffnet den RS232-Kanal mit 1 Stopp-Bit, 300 Bit/s und 7 Bit Wortlänge, außerdem über 3-Leitungs-Handshake, Voll-Duplex und ohne Parität.

Zeile 20 will den Eingabespeicher auslesen, der aber noch leer ist. Der resultierende Nullstring interessiert uns nicht, aber die Schnittstelle signalisiert über Handshake, daß wir bereit sind, Daten zu übernehmen.

Zeile 30 fragt inzwischen die Tastatur ab. Wenn eine Taste gedrückt worden ist, schiebt Zeile 40 das Zeichen in den Ausgabespeicher und druckt es nochmal auf dem Bildschirm aus.

Zeile 50 liest wieder den Eingabespeicher aus. Falls inzwischen Daten über die Schnittstelle gekommen sind, druckt Zeile 60 das erste Zeichen auf den Bildschirm.

Zeile 70 ordnet der Variablen PR(üfung) den Inhalt des Statusregisters 663 zu. Ist kein Fehler aufgetreten (PR=0) oder ist der Eingabespeicher immer noch leer (PR=8), dann springt Zeile 80 zurück zur Tastaturabfrage, und der Zyklus läuft weiter. Ist aber ein Fehler aufgetreten, wird dieser ab Zeile 100 geprüft und ausgedruckt.

Adresse 665 und 666 ($299 und $29A)

RS232 - Zeit, die zum Übertragen eines Bits gebraucht wird

Sobald ein RS232-Kanal eröffnet worden ist, berechnet das Betriebssystem einen Wert, der die Zeitdauer eines Bits festlegt. Da die Übertragungsrate in Speicherzelle 659 einstellbar ist, hängt diese Bit-Dauer von der gewählten Übertragungsgeschwindigkeit ab. Die Bit-Dauer errechnet sich aus der Systemfrequenz (985,25 kHz) geteilt durch die Übertragungsgeschwindigkeit. Dieser Wert steht in Low-/High-Byte-Darstellung in diesen beiden Speicherzellen, von wo aus er vom Betriebssystem abgerufen wird.

Adresse 667 ($29B)

RS232-Index auf das Ende des Eingabepufferspeichers

Dieser Index wird verwendet, um Daten in den Eingabepufferspeicher zu schreiben. Wenn man ihn nämlich zum Inhalt der Speicherzelle 247/248 addiert, erhält man die Adresse des zuletzt in den Eingabepufferspeicher eingegebenen Bytes.

Adresse 668 ($29C)

RS232-Index auf den Anfang des Eingabepufferspeichers

Dieser Index wird verwendet, um Daten aus dem Eingabepufferspeicher auszulesen. Wenn man ihn nämlich zum Inhalt der Speicherzelle 247 und 248 addiert, erhält man die Adresse des ersten in den Eingabepufferspeicher eingegebenen Bytes.

Adresse 669 ($29D)

RS232-Index auf den Anfang des Ausgabepufferspeichers

Dieser Index wird verwendet, um Daten aus dem Ausgabepufferspeicher auszulesen. Wenn man ihn nämlich zum Inhalt der Speicherzelle 249 und 250 addiert, erhält man die Adresse des ersten in den Ausgabepufferspeicher eingegebenen Bytes.

Adresse 670 ($29E)

RS232-Index auf das Ende des Ausgabepufferspeichers

Dieser Index wird verwendet, um Daten in den Ausgabepufferspeicher zu schreiben. Wenn man ihn nämlich zum Inhalt der Speicherzelle 249 und 250 addiert, erhält man die Adresse des zuletzt in den Ausgabepufferspeicher eingegebenen Bytes.

Adresse 671 und 672 ($29F und $2A0)

Zwischenspeicher für den IRQ-Vektor während Kassetten-Ein-/Ausgabe

Die Routinen des Betriebssystems, die Daten auf, beziehungsweise von Kassette ein- und ausgeben, werden durch die Interrupt-Routine gesteuert. Diese Routine unterbricht normalerweise 60mal in der Sekunde alle Aktivitäten des Computers, um diverse »Hausaufgaben« (Uhr weiterschalten, STOP-Taste abfragen und so weiter) auszuführen. Bei Kassetten-Ein-/Ausgaben ist diese Interrupt-Routine jedoch abgeschaltet. Dies wird dadurch erreicht, daß der Vektor in Speicherzelle 788 und 789, der auf die Anfangsadresse der Interrupt-Routine zeigt, auf eine Adresse der Kassetten-Routine gesetzt wird. Um nach der Kassettenoperation weitermachen zu können, wird der »alte« Interrupt-Vektor in dieser Speicherzelle 671 und 672 gespeichert.

Adresse 673 ($2A1)

bei C 64: Flagge für RS232-Interrupt bei VC 20: frei verfügbar

Diese Speicherzelle enthält den Wert des Interrupt-Steuerregisters 56589, das die RS232-Schnittstelle steuert. Die Bedeutung der einzelnen Bits, wenn sie auf 1 gesetzt sind, zeigt Tabelle 15. Diese Flagge kann zu Steuerzwecken abgefragt werden. Um beispielsweise ein Programm warten zu lassen, bis der Ausgabepufferspeicher geleert ist, gibt man die Anweisung

```
100 IF (PEEK(673) AND 1)
THEN 100
```

die das Programm so lange aufhält, bis die Übertragung abgeschlossen und Bit 0 der Flagge gelöscht ist.

Die folgenden 4 Speicherzellen, nämlich 674 bis 678, werden nur vom C 64 benutzt. Beim VC 20 sind sie nicht belegt und können frei verwendet werden.

Adresse 674 ($2A2)

Indikator für das Steuerregister A des CIA #1

Mit CIA werden die beiden »Complex Interface Adapter« des C 64 bezeichnet. Das sind integrierte Schaltkreise, die Ein- und Ausgabeoperationen steuern. Jeder der beiden CIAs hat mehrere Register. Das Steuerregister A (Adresse 56334 beziehungsweise $DC0E) beeinflußt die Zählregister des CIA, die ihrerseits die Ein- und Ausgabe von Daten auf beziehungsweise von Kassetten steuern. Das Betriebssystem speichert zu diesem Zweck geeignete Bitmuster in der Speicherzelle 674 ab, die von da in das Steuerregister transferiert werden.

Adresse 675 ($2A3)

Speicher für das Interrupt-Steuerregister B des CIA #1

Ein weiteres Register (Adresse 56333 beziehungsweise $DC0D) ist für die Unterbrechungen (Interrupt) des Computers bei Ein- und Ausgaben zuständig.

In der Speicherzelle 675 werden Werte dieses Interruptregisters beim Lesen von der Kassette zwischengespeichert.

Adresse 676 ($2A4)

Zusatzspeicher für Steuerregister B des CIA #1

Derselbe Wert, der bei der Vorbereitung des Lesevorganges von der Kassette in die Speicherzelle 674 kommt, gelangt auch nach 676, von wo er zu einem späteren Zeitpunkt beim Lesen zu Vergleichszwecken herangezogen wird.

Adresse 677 ($2A5)

Zwischenspeicher für das Link-Byte während des Bildschirm-Scrollens

Das Betriebssystem enthält eine Routine, welche den Bildschirminhalt hochschiebt (scrollt), sobald eine leere Zeile eingeschoben wird. Das bedeutet, daß jedesmal die Angaben in den Link-Tabellen der Speicherzellen 217 bis 241 geändert werden müssen. In der Speicherzelle 677 wird nun das Link-Byte zwischengespeichert, während der obere Teil des Bildschirms hochgeschoben wird. Beim VC 20 gibt es diese Funktion übrigens auch. Sie wird durch die Speicherzelle 242 ausgefüllt.

Adresse 678 ($2A6)

Flagge für PAL oder NTSC

Im Gegensatz zum VC 20, der entweder fest auf die deutsche Fernsehnorm PAL oder aber auf die amerikanische Norm NTSC eingestellt ist, kann der C 64 beide Normen verkraften. Diese beiden Normen beziehen sich unter anderem auf die Anzahl der Zeilen und auf die Abtast-Geschwindigkeit des Lichtstrahls im Fernsehgerät oder im Monitor. Das Betriebssystem des C 64 überprüft gleich beim Einschalten, ob eine Rasterzeile 311 im angeschlossenen Sichtgerät vorhanden ist. Ist sie nicht vorhanden, muß alles auf die NTSC-Norm eingestellt werden, da diese nur 262 Rasterzeilen hat und mit einer internen Taktfrequenz von 14,3 MHz läuft. Ist eine Rasterzeile 311 vorhanden, wird auf PAL-Norm eingestellt mit einer Taktfrequenz von 17,7 MHz. Das Resultat dieses Tests wird in der Speicherzelle 678 gespeichert: als 0 für NTSC und 1 für PAL.

Bit 0	(Bitwert 1)	Daten werden gesendet
Bit 1	(Bitwert 2)	Daten werden empfangen
Bit 4	(Bitwert 16)	Schnittstelle wartet auf Daten vom Sender

Tabelle 15. Flagge für RS232-Interrupt

Adresse 679 bis 767 ($2A7 bis $2FF)

nicht belegt

Diese 89 Byte sind frei und können für alle möglichen Programme und Anwendungen verwendet werden. Beim VC 20 stehen sogar 95 Byte zur Verfügung, da der freie Bereich ja schon ab Speicherzelle 673 beginnt. Dieser Speicherbereich hat den Vorteil, daß er - wie der Kassettenpuffer ja auch - von Basic nicht gestört wird. Er kann also für kleinere Maschinenprogramme oder auch für Sprite-Blöcke verwendet werden. Gegenüber dem Kassettenpuffer hat dieser Bereich den Vorteil, daß er durch Kassettenoperationen nicht gestört wird.

Die nächsten 12 Speicherzellen enthalten 6 Vektoren, deren Bedeutung bei der Übersetzung von Basic-Programmen im Texteinschub Nr. 31 »Indirekte Sprung-Vektoren« näher erklärt wird.

Adresse 768 und 769 ($300 und $301)

Vektor auf die Ausgabe von Fehler-Meldungen (ERROR)

Dieser Vektor zeigt auf die Anfangsadresse der Basic-Routine, welche für die leidigen Fehlermeldungen zuständig ist. Beim C 64 zeigt der Vektor auf 58251 ($E38B), beim VC 20 auf 50234 ($C438).

Diese Routine verwendet eine Tabelle im Basic-Übersetzer, in der alle Fehlermeldungen gespeichert sind. Sie liegt im Speicherbereich 41374 bis 41767 (beim VC 20 49566 bis 49959). Die Routine verwendet den Inhalt des X-Registers (siehe Speicherzelle 781), um die entsprechende Fehlermeldung ganz einfach durch Abzählen der Reihenfolge aus der Tabelle auszulesen und auf dem Bildschirm anzuzeigen.

Ein Verbiegen dieses Vektors ist für zwei Anwendungsfälle sinnvoll. Man kann die Fehlermeldung abschalten, um zu prüfen, ob ein bestimmtes Peripherie-Gerät, zum Beispiel das Floppylaufwerk, angeschlossen beziehungsweise eingeschaltet ist. Die Fehlermeldung ist abschaltbar mit POKE 768,61. Wieder eingeschaltet wird sie mit POKE 768,139. Ein Anwendungsbeispiel habe ich bereits im Texteinschub Nr. 14 »ST-atus« gebracht.

Die zweite Anwendung einer Verbiegung zielt auf eine Übersetzung der Fehlermeldungen. Wem der vorgegebene englische - und manchmal nicht gerade einleuchtende - Text der Fehlermeldungen nicht gefällt, kann den Vektor auf einen Speicherbereich legen, in dem er seine speziellen deutschen Fehlermeldungen speichert. Eine genaue Kenntnis der Fehlermeldungsroutine ist dazu allerdings erforderlich.

Adresse 770 und 771 ($302 und $303)

Vektor auf die Hauptroutine zur Ausführung von Basic-Befehlen

Dieser Vektor zeigt auf die Adresse 42115 ($A483), beim VC 20 auf 50307 ($C483). Die dort beginnende Routine steuert den Direkt-Modus, indem sie entweder direkt eingegebene Befehle ausführt oder mit Zeilennummer eingegebene Anweisungen speichert.

Adresse 772 und 773 ($304 und $305)

Vektor auf die Basic-Routine, die ASCII-Text in Token umwandelt

Dieser Vektor zeigt auf 42364 ($A57C), beim VC 20 auf 50556 ($C57C). Dort beginnt eine Routine, die nach dem Drücken der RETURN-Taste alle Anweisungen der damit eingegebenen Zeile absucht und Text beziehungsweise Wörter, die nicht zwischen Gänsefüßen stehen, als Basic-Befehle interpretiert und sie dann in sogenannte »Token« umwandelt. Token sind Codezahlen, die im Computer anstelle von Textbefehlen verwendet werden. Sie sind im Texteinschub Nr. 32 »Die Kurzschrift von Basic« näher beschrieben.

Dieser Vektor kann verbogen werden, um zusätzliche Basic-Befehle zu erfinden und in das Betriebssystem einzubauen.

Adresse 774 und 775 ($306 und $307)

Vektor auf die Basic-Routine, die Token in ASCII-Werte zurückwandelt (LIST)

Dieser Vektor zeigt auf die Adresse 42778 ($A71A), beim VC 20 auf 50970 ($C71A). Dort beginnt eine Routine, die Token wieder in LISTbaren Text umwandelt. Sie steht nicht allein, sondern wird als Unterprogramm von der LIST-Routine verwendet.

Falls ein Programmierer spezielle zusätzliche Basic-Befehle erfunden hat, kann er durch Verbiegen dieses Vektors seine eigenen Token lesbar ausLISTen.

Man kann auch durch eine entsprechende Verbiegung erreichen, daß die LIST-Routine nicht angesprungen werden kann, was gleichbedeutend ist mit einer LIST-Sperre. Das ist aber wohl nur sinnvoll bei einem Autostart-Programm.

Besser finde ich da ein kleines Programm, das J.Pellechi in der Zeitschrift RUN, Ausgabe 6/85 (Seite 10), angegeben hat:

```
10 FOR J=679 TO 688
20 READ K
30 POKE J,K
40 NEXT J
50 POKE 774,167:POKE 775,2
60 NEW
70 DATA 72,173,141
80 DATA 2,208,251,104
90 DATA 76,26,167
```

Beim VC 20 ist nur die Zeile 90 verschieden:

```
90 DATA 76,26,199
```

In den freien Speicherbereich ab Speicherzelle 679 wird ein kleines Maschinenprogramm gePOKEt, das in den DATA-Zeilen 70 bis 90 steht. In Zeile 50 steht der für unser Beispiel entscheidende Befehl: Der Vektor in 774 und 775 wird nach der Adresse 679 verbogen. Dadurch springt die LIST-Routine immer zuerst auf die Adresse 679, in der sie das kleine Maschinenprogramm findet.

Disassembliert schaut das so aus:

```
02A7 48       PHA
02A8 AD 8D 02 LDA 028D
02AB D0 FB    BNE 02A8
02AD 68       PLA
02AE 4C 1A A7 JMP A71A
```

Zuerst wird der Akkumulator mit dem Inhalt der Speicherzelle 653 ($28D) geladen. Dort steht bekanntlich eine Zahl von 1 bis 7, je nachdem, ob die SHIFT-, CTRL- oder Commodore-Taste gedrückt ist. Ist dies der Fall, springt das Programm auf die Adresse 680 zurück und bildet so eine Dauerschleife, bis die Taste wieder losgelassen wird. Erst dann geht es weiter mit der ursprünglichen Zieladresse des Vektors in 774 und 775, nämlich $A71A (42778) beziehungsweise $C71A (50970) beim VC 20.

Auf diese Weise können Sie das LISTen eines Programms mit einer der drei genannten Tasten anhalten.

Adresse 776 und 777 ($308 und $309)

Vektor auf die Basic-Routine, die den nächsten Befehl liest und ausführt

Dieser Vektor zeigt auf die Adresse 42980 ($A7E4), beim VC 20 auf 51172 ($C7E4). Diese Routine prüft das nächste Token, ob es gültig ist. Wenn der ASCII-Wert des Token kleiner als 128 ist, wird er als Zeichen einer Variablen angesehen, und das System springt auf die LET-Routine. Das erklärt, warum zur Definition einer Variablen der LET-Befehl auch weggelassen werden kann.

Durch Verbiegen dieses Vektors kann zum Beispiel eine Trace-Routine gebaut werden, welche zuerst die Nummer der Zeile ausdruckt, die gerade ausgeführt wird, bevor sie auf die ursprüngliche Zieladresse des Vektors zurückkehrt.

Adresse 778 und 779 ($30A und $30B)

Vektor auf die Basic-Routine, die einen numerischen Ausdruck in eine Gleitkommazahl umwandelt

Dieser Vektor zeigt auf 44675 ($AE83), beim VC 20 auf 52867 ($CE83). Hier beginnt eine Routine, die einen einzelnen numerischen Wert, wenn er Teil eines Ausdrucks ist, von seinem ASCII-Wert in eine Gleitkomma-Zahl umwandelt.

Ist der Ausdruck eine Konstante, wird diese Umwandlung durchgeführt.

Ist der Ausdruck eine Variable, wird ihr Zahlenwert aus dem Variablenspeicher geholt.

Ist der Ausdruck die Zahl »pi«, wird der Zahlenwert für »pi« in den Gleitkomma-Akkumulator gebracht.

Speicher für die Register des Mikroprozessors

Der SYS-Befehl holt aus den nächsten vier Speicherzellen alle notwendigen Parameter, die für ein mit SYS zu startendes Maschinenprogramm notwendig sind. Er speichert sie in die vier Register des Mikroprozessors 6510 (beim VC 20 heißt er 6502). Es sind dies:

- der Akkumulator
- das X-Register

- das Y-Register
- das P-(Status-)Register

Die Bedeutung der Register ist im Assembler-Kurs erklärt worden.

Normalerweise funktioniert der SYS-Befehl nur, wenn vorher schon alle Parameter des aufgerufenen Maschinenprogramms richtig vorhanden sind, was meistens nicht der Fall ist.

So können Sie zum Beispiel mit Aufrufen der Load-Routine durch SYS 62622 nichts ausrichten, weil die für LOAD erforderlichen Parameter, nämlich Gerätenummer, File-Namen, Anfangs- und Endadresse, nicht festgelegt sind.

Wie dies mit Hilfe der vier folgenden Register-Speicherzellen erreichbar ist, hat Rolf Zweifel schon in der Ausgabe 7/84, Seite 131 erklärt. Weil das aber schon lange her ist und weil es hier so schön in den Kurs paßt, wiederhole ich dieses Thema im Texteinschub Nr.33 »Der vorbereitete SYS-Befehl«.

Adresse 780 ($30C)

Speicher für den Akkumulator

Adresse 781 ($30D)

Speicher für das X-Register

Adresse 782 ($30E)

Speicher für das Y-Register

Adresse 783 ($30F)

Speicher für das Statusregister

Die nächsten drei Speicherzellen 784 bis 786 sind beim VC 20 nicht belegt. Beim C 64 entsprechen sie den Adressen 0 bis 2 des VC 20.

Adresse 784 bis 786 ($310 bis $312)

Sprungbefehl und wählbare Sprungadresse des USR-Befehls

Mit dem Basic-Befehl USR wird bekanntlich ein Maschinenprogramm gestartet. Diese drei Speicherzellen werden bei der Abwicklung von USR verwendet. In ihnen muß der Anwender des USR-Befehls die Zieladresse in Low-/High-Byte-Darstellung angeben, ab der das Maschinenprogramm im Speicher steht.

Dieser Vorgang ist bereits behandelt worden bei den Speicherzellen 0 bis 2 des VC 20, die ja genau den Speicherzellen 784 bis 786 des C 64 entsprechen.

Speziell für den C 64 ist der USR-Befehl noch einmal behandelt, und zwar im Texteinschub Nr. 34 »Das Mauerblümchen USR«.

Adresse 787 ($313)

beim C 64 und VC 20 nicht belegt

Während dieses freie Byte des C 64 nicht viel nutzt, haben VC 20-Besitzer immerhin vier aufeinanderfolgende freie Byte für eigene Vektoren und andere zwischenzuspeichernde Werte zur Verfügung, die nie in Gefahr geraten, von einem Basic-Programm überschrieben zu werden.

Texteinschub Nr. 31
Indirekte Sprung-Vektoren

Mit »Vektor« wird ein Adressenpaar bezeichnet, dessen Inhalt in der Low-/High-Byte-Darstellung wiederum eine Adresse angibt, ab der ein Maschinenprogramm beginnt.

Wenn man nun mit dem Maschinencode-Befehl JMP (jump) auf die Adresse springt, die der Vektor angibt, läuft das Maschinenprogramm ab dieser Adresse los.

Bekanntlich stehen im nicht veränderbaren ROM-Speicher viele Unterprogramme (Routinen) des Basic-Übersetzers und des Betriebssystems, die auch für andere Programme verwendbar sind. Commodore hat nun die brillante Idee gehabt, mehrere dieser Routinen herauszusuchen und ihre Anfangsadressen zur leichten Verwendung benutzerfreundlich in einer Tabelle zusammenzustellen, wo sie mit dem schon genannten Sprungbefehl »angewählt« werden können.

Diese Tabelle ist deshalb interessant, weil die Anfangsadressen der Routinen bei den einzelnen Commodore-Computern verschieden sind, obwohl sie eigentlich fast identische Übersetzer und Betriebssysteme haben. So beginnt zum Beispiel die LOAD-Routine des C 64 ab Adresse 62622 ($F49E), beim VC 20 aber ab Adresse 62786 ($F542).

Um zu erreichen, daß Programme, die diese Routinen benutzen, trotzdem von einem Commodore-Computer auf einen anderen übertragbar sind, hat Commodore diese Tabelle geschaffen, welche den Sprung auf diese Routinen unabhängig vom Computertyp macht.

Sie liegt (bei allen Commodore-Typen) im Bereich 768 bis 779 für Routinen des Basic-Übersetzers und 788 bis 819 für Routinen des Betriebssystems.

Diesen Zusammenhang zeige ich aber besser an einem Beispiel:

Der Vektor auf die LOAD-Routine hat die Adresse 816/817.

Wir schauen nach, was dort steht:

```
PRINT PEEK(816) PEEK(817)
```

Wir erhalten beim C 64 die Zahlen 158 und 244.

Beim VC 20 lautet das Ergebnis 66 und 245

Beide Zahlenpaare werden nach der üblichen Low-/High-Byte Methode umgerechnet:

158 + 256 * 244 ergibt 62622.

66 + 256 * 245 ergibt 62786.

Das sind aber genau die weiter oben schon genannten Anfangsadressen der LOAD-Routine im Betriebssystem. Mit einem Sprung auf 816 und 817 landet ein Maschinenprogramm also immer zwangsläufig auf der LOAD-Routine. Kenner wissen, daß ein derartiger Sprung »indirekt« sein muß, also mit dem Befehls-Code $6C, der nicht auf die angegebene Adresse, sondern auf die in ihr enthaltene Zieladresse springt.

Diese indirekte Sprungmethode hat außer der schon erwähnten Unabhängigkeit vom Computertyp noch einen weiteren Vorteil:

Da der Vektor, der auf die Zieladresse zeigt, im RAM-Speicher liegt, kann er verändert werden. Das bedeutet, daß dem Programmierer die Möglichkeit geboten wird, in ursprünglich »festgefrorene« Routinen des Übersetzers (Interpreter) und des Betriebssystems beliebige Änderungen und Varianten einzubauen. Ich will Ihnen das mit einem zwar nutzlosen, aber dennoch verblüffenden Beispiel zeigen. Bekanntlich meldet sich der Computer nach dem Befehl LOAD mit der Anweisung »PRESS PLAY ON TAPE«, weil der Vektor in 816 und 817 auf die Routine zeigt, die den LOAD-Befehl ausführt.

Jetzt verbiegen wir den Vektor so, daß er auf die SAVE-Routine zeigt. Diese Routine beginnt ab Speicherzelle ($E156) - beim VC 20 ab 57683 ($E153). Diese Adresse POKEn wir in Low-/High-Byte-Darstellung nach 816 und 817.

```
POKE 816,86:POKE 817,225 (VC 20:POKE 816,83)
```

Wenn Sie jetzt LOAD eingeben, meldet sich der Computer mit »PRESS RECORD AND PLAY ON TAPE«, der Anweisung für SAVE.

Weitere Beispiele dafür finden Sie im Kurs bei den entsprechenden Speicherzellen. Diese Art der Modifikation entspricht der oft genannten Wedge-Methode, die auch ich in diesem Kurs erwähnt habe, und zwar bei der Besprechung der Speicherzellen 115 bis 138. Aber der Einsatz der indirekten Sprung-Vektoren ist halt viel eleganter.

Texteinschub Nr. 32
Die Kurzschrift von Basic

Immer wenn Sie eine Anweisung auf den Bildschirm schreiben, sei es als Programmzeile, sei es als Direkt-Eingabe, wird sie in den Tastaturpuffer gebracht. Sobald die Eingabe mit der RETURN-Taste abgeschlossen wird, werden die Anweisungen dieser Zeile entweder in den Programmspeicher gebracht oder aber direkt ausgeführt. Beides geschieht allerdings nicht sofort, da der Computer ja bekanntlich intern nicht mit Buchstaben und Dezimalziffern, sondern mit besonderen Codezahlen arbeitet. Deshalb wird der Text einer Zeile zuerst in diese Codewerte umgewandelt und in eine besondere Reihenfolge gebracht. Ziffern, Zeichen und Texte, die innerhalb von Gänsefüßen stehen,

werden mit ihrem ASCII-Code abgespeichert. Basic-Befehle und Basic-Funktionen werden in Zahlen umgewandelt. Diese heißen in der Fachsprache »TOKEN«.

Der Befehl PRINT wird also nicht als Folge von 5 ASCII-Werten, sondern als einzelnes Byte mit dem Wert 153 gespeichert. Da meines Wissens die Liste der Token nur im Programmierhandbuch des VC 20 und im 64'er, Ausgabe 9/84 im Kurs von Christoph Sauer veröffentlicht ist, gebe ich in der weiter unten folgenden Tabelle alle Werte noch einmal an.

Bemerkenswert ist, daß für GET # kein eigener Token vorhanden ist, dafür aber einer (203) für einen Befehl, den es in den Handbüchern nicht gibt, nämlich GO. Das erklärt übrigens, warum der Befehl GOTO auch in der Form GO TO geschrieben werden darf. Die Routine, welche Basic-Befehle ausführt, erkennt nämlich die beiden Token an und kombiniert sie miteinander.

Interessant ist auch, daß die Befehle TAB und SPC in ihren Token gleich die linke Klammer mit einschließen. Nach dieser Tabelle sieht eine im Programmspeicher gespeicherte Zeile so aus:

```
10 IF A=5THEN PRINT TAB(X)
```

Link	10	0	139	65	178	53	167	32	153	32	163	88	41	0
	Zeil.-Nr.		IF	A	=	5	THEN		PRINT		TAB(	X	)	

Diese Zahlen können Sie selbst überprüfen, indem Sie diese Zeile eingeben und dann den Anfang des Programmspeichers sichtbar machen:

```
FOR J=1 TO 20:PRINT PEEK(2048)+J);:NEXT
```

Beim VC 20 müssen Sie den von der Speichererweiterung abhängigen Anfang des Programmspeichers einsetzen. Sie werden dieselbe Zahlenreihe wie oben erhalten.

Die Technik, in einem Programm direkt die Token anstelle von Basic-Befehlswörtern zu verwenden, bieten dem Programmierer in Maschinensprache eine gute Möglichkeit, Speicherplatz zu sparen. Das kann insbesondere bei großen Textprogrammen, wie zum Beispiel bei Adventure-Spielen, nützlich sein. Der Vollständigkeit halber muß ich noch erwähnen, daß die Token bei dem LIST-Befehl wieder in ihre ursprüngliche Textform zurückgewandelt werden. Die Vektoren für die Wandel- beziehungsweise Rückwandel-Routinen stehen in 772 und 773 und 774 und 775.

```
128 END          154 CONT       180 SGN
129 FOR          155 LIST       181 INT
130 NEXT         156 CLR        182 ABS
131 DATA         157 CMD        183 USR
132 INPUT#       158 SYS        184 FRE
133 INPUT        159 OPEN       185 POS
134 DIM          160 CLOSE      186 SQR
135 READ         161 GET        187 RND
136 LET          162 NEW        188 LOG
137 GOTO         163 TAB(       189 EXP
138 RUN          164 TO         190 COS
139 IF           165 FN         191 SIN
140 RESTORE      166 SPC(       192 TAN
141 GOSUB        167 THEN       193 ATN
142 RETURN       168 NOT        194 PEEK
143 REM          169 STEP       195 LEN
144 STOP         170 +          196 STR$
145 ON           171 -          197 VAL
146 WAIT         172 *          198 ASC
147 LOAD         173 /          199 CHR$
148 SAVE         174 ↑          200 LEFT$
149 VERIFY       175 AND        201 RIGHT$
150 DEF          176 OR         202 MID$
151 POKE         177 größer     203 GO
152 PRINT#       178 =
153 PRINT        179 kleiner
```

Texteinschub Nr. 33
Der vorbereitete SYS-Befehl

Programme, die in Maschinensprache geschrieben sind, können von einem Basic-Programm aus mit dem SYS-Befehl angewählt und ausgeführt werden.

Im Prinzip gilt das auch für Routinen des Basic-Übersetzers (Interpreter) und des Betriebssystems, die fest im ROM-Speicher untergebracht sind.

Ein Beispiel dafür ist SYS 58260, der Sprung auf den Kaltstart - beim VC 20 ist es SYS 58232, der den Computer in die Ausgangslage zurücksetzt.

Die meisten dieser Routinen benötigen jedoch verschiedene Angaben - man nennt sie auch Parameter - die vor der Ausführung des SYS-Befehls richtig eingestellt sein müssen.

Die LOAD-Routine zum Beispiel, die ab Speicherzelle 62622 ($F49E) - beim VC 20 ab 62793 ($F549) - beginnt, können wir mit dem SYS 62622 nicht starten. Es fehlen die Angaben über Geräte-Nummer (8 für Floppy, 1 für Band), File-Namen, sowie Anfangs- und Endadresse. Diese Parameter werden normalerweise nach dem Befehl LOAD von der Routine des Interpreters, die den LOAD-Befehl übersetzt, eingegeben. Wir geben ja nicht einfach LOAD ein, wenn wir ein Programm mit dem Namen »Test« auf Diskette speichern wollen, sondern wir schreiben

```
LOAD "TEST",8.
```

Auch wenn wir nur LOAD eintippen, werden vom Übersetzer Parameter gesetzt, nämlich »namenlos« und 1 für Bandgerät. Ich hoffe, Ihnen ist geläufig, daß beim Weglassen aller Angaben der Übersetzer immer Kassettenoperationen durchführt. Natürlich können wir uns das anschauen:

Die Routine des Übersetzers für den Basic-Befehl LOAD beginnt an Speicherzelle 57704 ($E168), beim VC 20 bei 57700 ($E164).

Mit SYS 57704 springen wir dorthin - und in der Tat, wir erhalten auf dem Bildschirm »PRESS PLAY ON TAPE«. Aber ein Programm auf diese Weise von der Floppy zu LOADen, gelingt uns nicht, es sei denn, wir können die fehlenden Parameter von Hand eingeben.

Genau das aber können wir, weil der SYS-Befehl sich diese Parameter aus den Speicherzellen 780 bis 783 holt und in die vier Register des Mikroprozessors schreibt.

780 ist die Adresse des Akkumulators
781 ist die Adresse des X-Registers
782 ist die Adresse des Y-Registers
783 ist die Adresse des Status-Registers.

Die Behandlung von A, X und Y ist unkompliziert, wie wir gleich sehen werden.

Das Status-Register, manchmal auch P-Register genannt, ist nicht so einfach zu verwenden, da es nicht Zahlenwerte, sondern Flaggen (Bitmuster) enthält. Im einzelnen bedeuten:

BIT Nr.	WERT	FLAGGE	ABKÜRZUNG
0	1	Übertrag	C(arry)
1	2	NULL	Z(ero)
2	4	Unterbrechung	I(nterrupt)
3	8	Dezimal	D
4	16	Abbruch	B(reak)
5	32	nicht benutzt	
6	64	Überlauf	V
7	128	Vorzeichen	N(egativ)

Um eine der Flaggen des Status-Registers zu löschen, empfiehlt es sich, das ganze Register mit `POKE 783,0` zu löschen. Umgekehrt muß man beim Setzen der Bits sehr aufpassen wegen der Unterbrechungsflagge I. Eine 1 in I entspricht dem Maschinenbefehl SEI, der alle Interrupts ausschaltet, auch die der Tastatur-Abfrage, was natürlich sehr störend sein kann! Um alle Flaggen außer der Unterbrechungsflagge I zu setzen, muß `POKE 783,247` eingegeben werden.

So, jetzt wird es Zeit für ein Beispiel, wie vor dem SYS-Befehl Parameter eingegeben werden können. In der Literatur wird immer das Beispiel gewählt, den Cursor auf eine bestimmte Posi-

tion zu setzen, beziehungsweise seine Position abzufragen. Dazu gibt es eine Routine, die bei beiden Computern ab Speicherzelle 65520 ($FFF0) beginnt.

Sie nimmt die Zahl, die im X-Register steht, und verwendet sie als Zeilennummer; die Zahl des Y-Registers nimmt sie als Spaltennummer, setzt dann den Cursor an diese Stelle und bringt die beiden Werte in die Speicherzellen 209 und 210 und 211.

Unser Beispiel hat die Aufgabe, den Cursor in die vierte Spalte der siebten Zeile zu setzen, dort das Dollar-Zeichen hinzuschreiben und es rot zu färben.

```
 5 PRINT CHR$(147)
10 POKE 783,0
20 POKE 781,6
30 POKE 782,3
40 SYS 65520
```

Nach Löschen des Bildschirms werden zuerst alle Flaggen des Statusregisters gelöscht (Zeile 5). Dann kommt die Zeilennummer in das X-Register (Zeile 10) und die Spaltennummer in das Y-Register (Zeile 30). Nach dem Eingeben dieser Parameter können wir mit SYS auf die Routine springen.

```
50 ZEILE=PEEK(209)+256*PEEK(210)
60 ADRESSE = ZEILE + PEEK(211)
70 POKE ADRESSE,36
```

In Speicherzellen 209/210 können wir jetzt (zur Übung) die Zeilennummer ablesen. Die Adresse der Cursorposition im Bildschirmspeicher erhalten wir durch die Addition der Zeilennummer mit dem Inhalt der Speicherzelle 211. Dorthin POKEn wir den Bildschirmcode des Dollarzeichens, nämlich 36 (Zeile 70).

```
 80 SYS 59940
 90 FARBE=PEEK(243)+256*PEEK(244)
100 POKE FARBE+PEEK(211),2
```

Für das Färben des Dollarzeichens verwenden wir eine weitere Routine des Betriebssystems, die ab 59940 - beim VC 20 ab 60082 - beginnt. Sie ermittelt die Zeilenposition des Cursors im Farbspeicher und bringt diesen Wert in die Speicherzellen 243 und 244, wo wir ihn abfragen können (Zeile 90). Die Adresse der Cursorposition im Farbspeicher setzt sich aus diesem Wert plus der Spaltennummer zusammen, die wir wieder der Speicherzelle 211 entnehmen. Auf diesen Platz POKEn wir den Farbcode 2 für rot (Zeile 100). So leicht ist das, wenn man die Routinen und die Aufgaben der Speicherzellen kennt.

Die letzteren lernen Sie in diesem Kurs. Die Beschreibung und Anwendung der Routinen muß, wie schon öfters erwähnt, einem eigenen Kurs vorbehalten bleiben.

Texteinschub Nr. 34
Das Mauerblümchen USR

Hand aufs Herz: Haben Sie den USR-Befehl schon einmal benutzt? Ohne Zweifel gehört er zu den Mauerblümchen von Basic, obwohl sein Name - eine Abkürzung von USER (Verwender) - eigentlich genügend Anreiz bieten müßte. Da er ohne die Speicherzellen 784 bis 786 nicht auskommt, ist dieser Teil des Kurses eine gute Gelegenheit, ihn Ihnen näherzubringen.

USR hat im Grunde genommen dieselbe Funktion wie SYS. Er springt nämlich aus einem Basic-Programm direkt in ein Maschinen-Programm, arbeitet dieses so lange ab, bis er den Befehl RTS findet. RTS entspricht dem Basic-Befehl RETURN und springt in das Basic-Programm zurück.

Bei SYS steht die Sprungadresse gleich hinter dem Befehl.

Bei USR muß die Sprungadresse zuerst in die Speicherzellen 785 und 786 gePOKEt werden (beim VC 20 in 1 und 2).

Beispiel: Sprung auf 56524 ($DCCC)

mit SYS: `SYS 56524`

mit USR:
```
POKE 785,204 (204+256*220=56524) POKE 786,220
X=USR(Y)
```

Kein Wunder, daß USR selten verwendet wird - ist er doch durch das POKEn der Sprungadresse in Low-/High-Byte-Darstellung aufgebläht.

Das ist aber nicht unnütz, weil USR mehr Fähigkeiten hat als SYS. Im Hinblick auf die im anderen Texteinschub Nr. 33 »Der vorbereitete SYS-Befehl« aufgezeigten Möglichkeiten des SYS-Befehls sollte ich besser sagen: USR hat andere Fähigkeiten als SYS.

USR ist eine Mischung von SYS und FN. Letzterer ist der Basic-Befehl zur Definition selbst erfundener Funktionen. Bei USR allerdings wird die Funktion als Unterprogramm in Maschinensprache geschrieben, auf die dann wie gesagt der USR-Befehl zur Ausführung springt. Der Pfiff dabei ist aber, daß Zahlenwerte in das Maschinenprogramm mitgenommen beziehungsweise Resultate aus ihm herausgeholt werden können.

Wie läuft das ab: Das Argument Y, das in der Klammer hinter dem Befehl steht, wird zuerst in den Gleitkomma-Akkumulator Nr. 1 (FAC 1) in den Speicherzellen 97 bis 102 gebracht. Als Gleitkommazahl wird es vom angesprungenen Maschinenprogramm weiterverarbeitet. Das Resultat kommt dann wieder in den FAC 1 und steht als Wert von X zur Verfügung.

Das Argument Y kann übrigens auch ein komplexer Ausdruck sein, zum Beispiel: `X=USR(PEEK(A)+256*PEEK(B))`

Ich möchte das an einem kleinen Beispiel demonstrieren.

Statt allerdings ein Maschinenprogramm selbst zu schreiben, verwende beziehungsweise springe ich auf eine Routine des Betriebssystems, die den Inhalt des FAC 1 für mathematische Operationen verwendet.

Als geeignete mathematische Operation habe ich die Routine für die Funktion INT gewählt, die im C 64 ab der Adresse 48332 ($BCCC) beginnt, beim VC 20 ab 56524 ($DCCC).

Zuerst definieren wir einen Wert für die Variable Y, der in die INT-Routine gebracht werden soll:

```
10 Y=14,35
```

Dann bestimmen wir die Sprungadresse für den USR-Befehl. Dazu teilen wir die Adresse 48332 auf in ein Low-Byte = 204 und ein High-Byte = 188. Bei der Startadresse 56524 von INT im VC 20 sind dies 204 und 220. Diese POKEn wir nach 785 und 786 beziehungsweise nach 1 und 2:

```
20 POKE 785,204
30 POKE 786,188
```

Jetzt folgt nur noch der USR-Befehl selbst und das Ausdrucken des Resultats.

```
40 X=USR(Y)
50 PRINT X
```

Beim VC 20 lauten die Zeilen 20 und 30:

```
20 POKE 1,204
30 POKE 2,220
```

Nach RUN erhalten wir das Resultat 14, wie das Gesetz für INT es befiehlt.

Sie können zur Übung statt INT auch COS verwenden, indem Sie auf die Adresse 57938 ($E264) beziehungsweise beim VC 20 auf 57935 ($E261) springen. Der Vergleich mit dem Befehl COS Y muß dasselbe Ergebnis bringen. Wer hat übrigens gemerkt, daß wir überhaupt nichts mit der Speicherzelle 784 (beziehungsweise 0) gemacht haben, obwohl sie doch angeblich am USR-Befehl beteiligt ist?

Sie ist es wirklich, doch ohne unser Zutun. In diese Adresse wird beim Einschalten des Computers die Zahl 76 ($4C) geschrieben. Das ist der Code für den Maschinenbefehl »JMP« (JUMP), der dieselbe Wirkung hat wie GOSUB.

Bei Ausführung von USR springt nämlich die entsprechende Routine zuerst auf die Speicherzelle 784 (beziehungsweise 0), findet dort den Sprungbefehl und in den beiden nachfolgenden Speicherzellen 785 und 786 (beziehungsweise 1 und 2) die Sprungadresse - und führt so den geplanten Sprung aus.

Ich finde, USR ist es wert, in Ihre Überlegungen mit einbezogen zu werden, besonders wenn Sie innerhalb Ihrer Basic-Programme extrem schnelle Unterprogramme in Maschinensprache eingebaut haben. Diese sind mit USR ganz elegant aufrufbar. Ich denke da zum Beispiel an eine Abfrage der Joysticks oder der Paddle.

Adresse 788 und 789 ($314 und $315)

Vektor auf die IRQ-Interrupt-Routine des Betriebssystems

Dieser Vektor zeigt auf die Adresse 59953 ($EA31) - beim VC 20 auf 60095 ($EABF). Ab hier beginnt die Routine des Betriebssystems, die den IRQ-Interrupt ausführt. Die Bedeutung der verschiedenen Interrupts (Unterbrechungen), ihre Auslöser und Abläufe sind als Übersicht im Texteinschub Nr. 35 »Dem Computer ins Wort fallen« dargestellt.

Die IRQ-Routine wird vom Timer A des Ein-/Ausgabe-Bausteins CIA #1 - beim VC 20 vom Timer 1 des Ein-/Ausgabe-Bausteins VIA #2 - ausgelöst, und zwar periodisch 60mal in jeder Sekunde. In der Programmpause werden die im Texteinschub beschriebenen »Haushaltsarbeiten« durchgeführt.

Dieser Vektor eignet sich hervorragend für eigene Programmierzwecke, da er durch das Verbiegen auf eine andere Adresse seine gleichmäßige und hochfrequente Wiederkehr nicht verliert. Mit seiner Hilfe können also eigene Maschinenprogramme 60mal in der Sekunde in ein Programm eingeschoben werden - eine Methode, die deswegen den englischen Namen »Wedge« = Keil, erhalten hat. Zwei Vorbedingungen sind allerdings dabei zu erfüllen.

1. Da ein IRQ mit Sicherheit während des Verbiegens auftritt, muß er vorher abgeschaltet werden. Den Schlüssel dazu bietet die Speicherzelle 56334, die mit 0 gePOKEt den Interrupt abschaltet und mit POKE 56334,1 ihn wieder zuläßt. Beim VC 20 ist dies POKE 37116,127 beziehungsweise POKE 37116,192. Aber Vorsicht!! Da während eines IRQ-Interrupts auch die Tastatur abgefragt wird, kann das Abschalten nur innerhalb eines Programms erfolgen - während der Abschaltung ist die Tastatur tot.

2. Am Ende eines »Wedge« muß der Sprung auf die alte IRQ-Adresse erfolgen, die ursprünglich in den Speicherzellen 788 bis 789 stand, damit - etwas verspätet zwar - die normalen Haushaltsarbeiten des IRQ nachgeholt werden können. Bei längeren Wedges wird daher die interne Uhr TI und TI$ etwas nachgehen.

Ich habe lange nach einem Beispiel gesucht. Ich kenne viele: Abfrage der Joysticks, Lautstärke von Tönen mit Funktionstasten steuern, von Basic unabhängige Laufschrift, um ein paar zu nennen. Aber alle haben einen ziemlich langen Maschinensprache-Teil. Ich bringe daher hier das kürzeste Beispiel, das ich kenne. Es stammt von Rügheimer und Spanik.

Das Programm verändert dauernd die Farbe des Bildschirmrahmens:

```
10 FOR K=679 TO 699
20 READ A
30 POKE K,A:NEXT
40 DATA 166,162,224,0,
   224,128,240
50 DATA 3,76,49,234,174,
   32,208
60 DATA 202,142,32,208,
   76,175,02
70 POKE 56334,0
80 POKE 788,167:
   POKE 789,2
90 POKE 56334,1
```

Dieses Programm gilt nur für den C 64; für den VC 20 müßte es entsprechend umgeschrieben werden.

Die Zeilen 10 bis 30 lesen das Maschinenprogramm, das in den DATA-Zeilen 40 bis 60 steht, in die Speicherzellen 679 bis 699. Diese stehen, wie wir das letzte Mal gesehen haben, zur freien Verfügung - und sind daher ideal geeignet, ein kleines Maschinenprogramm ungestört aufzunehmen.

In Zeile 70 wird der IRQ-Interrupt unterbrochen. Jetzt kommt der wichtige Teil: Zeile 80 verbiegt den IRQ-Vektor zur Speicherzelle 176 + 256 * 2 = 679. Zeile 90 schaltet schließlich den IRQ-Interrupt wieder ein.

Jetzt passiert also folgendes: Jedesmal, wenn der Timer A den Haushalt-IRQ auslöst, springt der Computer zuerst einmal auf das Maschinenprogramm ab Speicherzelle 679 und schaltet die Rahmenfarbe um. Dann erst springt der letzte Befehl des Maschinenprogramms auf die ursprüngliche IRQ-Adresse 59953 ($EA31), von der aus das Betriebssystem weitermacht, als sei nichts geschehen.

Für Kenner gebe ich noch das Assembler-Listing des Maschinenprogramms an:

```
,02A7 A6 A2    LDX A2
,02A9 E0 00    CPX #00
,02AB E0 80    CPX #80
,02AD F0 03    BEQ 02B2
,02AF 4C 31 EA JMP EA31
,02B2 AE 20 D0 LDX D020
,02B5 CA       DEX
,02B6 8E 20 D0 STX D020
,02B9 4c Af 02 JMP 02AF
```

Es gibt noch eine kleine, erwähnenswerte Anwendung. Wenn der Vektor nicht auf den Anfang der IRQ-Routine bei 59953, sondern auf 59956 - also drei Stellen weiter - zeigt, übergeht die IRQ-Routine den Teil, welcher die STOP-Taste abfragt und die TI/TI$-Uhr weiterschaltet, wodurch effektiv die STOP-Taste ausgeschaltet wird. Das geht ganz schnell mit POKE 788,52. Mit POKE 788,49 wird das wieder rückgängig gemacht. Beim VC 20 sind es die Werte POKE 788,194 oder POKE 788,191.

Adresse 790 und 791 ($316 und $317)

Vektor auf die BREAK-Interrupt-Routine des Betriebssystems

Diese Routine ist im Texteinschub Nr. 35 nicht erwähnt, weil sie ein Teil der NMI-Routine ist. Dieser Vektor zeigt auf die Adresse 65126 ($FE66) - beim VC 20 auf 65234 ($FED2). Die da beginnende Routine des Betriebssystems wird aufgerufen, wenn der Maschinenbefehl BRK ausgeführt wird. Er führt letztlich zu einem Warmstart, das heißt der Bildschirm wird gelöscht und der Cursor meldet sich mit READY. Diese Routine wird auch durch das gleichzeitige Drücken der STOP- und der RESTORE-Taste angestoßen.

Adresse 792 und 793 ($318 und $319)

Vektor auf die NMI-Routine des Betriebssystems.

Der NMI-Interrupt ist im Texteinschub Nr. 35 »Dem Computer ins Wort fallen« näher beschrieben. Der Vektor zeigt auf den Beginn dieser Routine ab Speicherzelle 65095 ($FE47) - beim VC 20 ab 65197 ($FEAD).

Sobald ein NMI-Interrupt auftritt, wird zuerst durch Setzen der Interrupt-Abschalt-Flagge (Interrupt Disable Flag) jede Unterbrechung durch den IRQ-Interrupt unterbunden. Dann wird geprüft, wer den NMI-Interrupt ausgelöst hat, und zwar in der Reihenfolge: RS232-Schnittstelle, RESTORE-Taste; eingestecktes Modul und schließlich die STOP-Taste. Die letztere dient zum Sichern der RESTORE-Taste. Nur wenn beide gemeinsam gedrückt werden, kommt die NMI-Unterbrechung durch die RESTORE-Taste zur Auswirkung.

Da die RESTORE-Taste fast als erste abgefragt wird, kann sie und ihre Kombination mit der STOP-Taste durch Verbiegen des Vektors in Speicherzelle 792 bis 793 abgeschaltet werden. Beim C 64 geht das mit POKE 792,193. Wieder eingeschaltet wird mit POKE 792,71. Beim VC 20 geht das mit POKE 792,91 beziehungsweise POKE 792,173. Natürlich können Spezialisten durch Verbiegen des Vektors auf andere Adressen ihre eigenen NMI-Routinen bauen.

Adresse 794 und 795 ($31A und $31B)

Vektor auf die OPEN-Routine des Betriebssystems

Die Routine beginnt ab Adresse 62282 ($F34A) - beim VC 20 ab 62474 ($FEAD). Diese Routine prüft, ob eine Datei (File) eröffnet werden kann. Das geht immer dann, wenn die File-Nummer nicht 0 ist und wenn weniger als 10 andere Dateien bereits eröffnet sind. Für die serielle Schnittstelle (Geräte-Nummer 4, 5, 8 bis 11) wird an das angewählte Gerät zuerst der Befehl »Listen« gegeben und dann die Sekundär-Adresse des OPEN-Befehls.

Beim Bandgerät (Geräte-Nummer 1) prüft die Routine den Tape Header einer sequentiellen Datei beziehungsweise schreibt einen Tape Header auf das Band.

Bei Anwahl der RS232-Schnittstelle (Geräte-Nummer 2) aktiviert die Routine einige Leitungen und reserviert je einen Ein- und Ausgabe-Pufferspeicher am oberen Ende des Basic-Programmspeichers.

Adresse 796 und 797 ($31C und $31D)

Vektor auf die CLOSE-Routine des Betriebssystems

Dieser Vektor zeigt auf die Adresse 62097 ($F291) - beim VC 20 auf 62282 ($F34A). Ab hier beginnt eine Routine, die beim CLOSE-Befehl zuerst prüft, ob die Datei-Nummer in der Tabelle der eröffneten Datei enthalten ist. Dann holt sie die dazugehörige Geräte-Nummer und Sekundär-Adresse und schließt den Kanal und die Datei.

Adresse 798 und 799 ($31E und $31F)

Vektor auf die CHKIN-Routine des Betriebssystems

Diese Routine beginnt ab Adresse 61966 ($F20E) - beim VC 20 ab 62151 ($F2C7). Sie eröffnet einen Datenkanal zur Übernahme von Daten von dem Gerät, das durch den OPEN-Befehl angegeben worden ist.

Adresse 800 und 801 ($320 und $321)

Vektor auf die CKOUT-Routine des Betriebssystems

Dieser Vektor zeigt auf die Adresse 62032 ($F250) - beim VC 20 auf 62217 ($F309). Dort beginnt die Routine, welche einen Datenkanal zur Abgabe von Daten an das im OPEN-Befehl angegebene Gerät aufmacht.

Adresse 802 und 803 ($322 und $323)

Vektor auf die CLRCHN-Routine des Betriebssystems

Der Name dieser Routine ist die Abkürzung für »clear channel«. Diese Routine, die ab Adresse 62259 ($F333) - beim VC 20 ab 62451 ($F3F3) - beginnt, setzt alle Kanäle in den Einschaltzustand zurück. Das heißt, das Eingabegerät ist die Tastatur, das Ausgabegerät ist der Bildschirm.

Adresse 804 und 805 ($324 und $325)

Vektor auf die CHRIN-Routine des Betriebssystems

Dieser Vektor zeigt auf die Adresse 61783 ($F157) - beim VC 20 auf 61966 ($F20E). Die hier beginnende Routine, deren Abkürzung »Character Input« bedeutet, holt das jeweils nächste Byte vom Eingabepuffer des angewählten Gerätes, sofern ein solcher eingerichtet ist (zum Beispiel Kassettenpuffer, RS232-Puffer).

Bei Eingabe von der Tastatur holt diese Routine so lange Bytes aus dem Tastaturpuffer und zeigt sie auf dem Bildschirm an, bis das Zeichen für ein ungeSHIFTetes RETURN auftritt. Erst dann gibt die Routine das erste Zeichen der logischen Zeile auf dem Bildschirm an den Basic-Übersetzer weiter.

Adresse 806 und 807 ($326 und $327)

Vektor auf die CHROUT-Routine des Betriebssystems

Die CHROUT-Routine entspricht der CHRIN-Routine in der anderen Richtung. Sie bedeutet »Character Output« und transferiert ein Byte, das im Akkumulator steht, in den Puffer des angewählten Ausgabegerätes. Sie beginnt ab Adresse 62898 ($F1CA), - beim VC 20 ab 62074 ($F27A).

Adresse 808 und 809 ($328 und $329)

Vektor auf die STOP-Routine des Betriebssystems

Der Vektor zeigt auf die Adresse 63213 ($F6ED) - beim VC 20 auf 63344 ($F770). Die dort beginnende Routine prüft, ob die STOP-Taste gedrückt ist. Durch Verbiegen dieses Vektors kann die STOP-Taste abgeschaltet werden. Beim C 64 geht dies mit `POKE 808,239`; wieder eingeschaltet wird die STOP-Taste mit `POKE 808,237`. Beim VC 20 sind die Werte `POKE 808,100` beziehungsweise `POKE 808,112`.

Adresse 810 und 811 ($32A und $32B)

Vektor auf die GETIN-Routine des Betriebssystems

Diese Routine ist fast identisch mit der CHRIN-Routine (siehe Speicherzellen 804 bis 805). Sie holt genauso Zeichen von angewählten Geräten in die Eingabepuffer. Der einzige und damit wichtigste Unterschied liegt in der Behandlung der Tastatur-Eingabe. Im Gegensatz zu CHRIN holt sie ein Byte aus dem Tastaturpuffer sofort in den Akkumulator. Der Vektor zeigt auf den Anfang der Routine ab Speicherzelle 61785 ($F13E) - beim VC 20 ab 61941 ($F1F5).

Adresse 812 und 813 ($32C und $32D)

Vektor auf die CLALL-Routine des Betriebssystems

CLALL ist die Abkürzung für Close ALL (Channels and Files). Diese Routine, die ab Adresse 62255 ($F32F) - beim VC 20 ab 62447 ($F3EF) - beginnt, setzt die Speicherzelle 152 auf 0 und schließt so zwangsläufig alle Dateien und Kanäle.

Adresse 814 und 815 ($32E und $32F)

Freier Vektor

Nach dem Einschalten zeigt dieser Vektor auf die BREAK-Routine, genauso wie der Vektor in Speicherzelle 790 und 791. Er ist ein Überbleibsel aus dem PET-Betriebssystem, das aber beim VC 20 und C 64 keine Rolle spielt. Hier können also eigene Vektoren definiert und eingesetzt werden.

Adresse 816 und 817 ($330 und $331)

Vektor auf die LOAD-Routine des Betriebssystems

Dieser Vektor zeigt auf die Adresse 62622 ($F49E) - beim VC 20 auf 62793 ($F549). Die dort beginnende Routine transferiert Daten von einem Eingabegerät direkt in den RAM-Speicher. Sie kann auch zum VERIFYen durch Vergleich der geladen mit den gespeicherten Daten verwendet werden.

Adresse 818 und 819 ($332 und $333)

Vektor auf die SAVE-Routine des Betriebssystems

Diese Routine ist das Gegenstück zur LOAD-Routine. Sie beginnt ab Adresse 62941 ($F5DD) - beim VC 20 ab 63109 ($F685).

Adresse 820 bis 827 ($334 bis $33B)

Freier Speicherbereich

Diese 8 Byte stehen zur freien Verfügung.

Adresse 828 bis 1019 ($33C bis $3FB)

Kassettenpuffer

Diese 192 Byte beherbergen den Kassettenpuffer. Der Name kennzeichnet diesen Speicherbereich als Zwischenspeicher für Ein- und Ausgabe-Operationen von und auf Band.

Dabei unterscheiden sich die normalen LOAD-, SAVE- und VERIFY-Befehle von den Datei-Befehlen INPUT#, GET# und PRINT#.

Bei LOAD, SAVE und VERIFY steht im Kassettenpuffer lediglich der Vorspann, der auf englisch »Tape Header« heißt. Die Funktion und Zusammensetzung des Tape Headers habe ich schon bei den Speicherzellen 183 bis 187 und im Texteinschub Nr. 20 »Tape Header« detailliert beschrieben. Die eigentlichen Daten berühren den Kassettenpuffer nicht, sondern werden direkt von und in den RAM-Speicher transferiert.

Bei GET#, INPUT# und PRINT# werden nicht nur der Tape Header, sondern auch alle Daten im Kassettenpuffer zwischengespeichert. Dieser blockweise Transport ist an den charakteristischen Unterbrechungen des Datasettenmotors leicht zu erkennen.

Der Kassettenpuffer kann durch Verbiegen der Zeiger in Speicherzelle 178 und 179 auf beliebige Plätze des Speichers, aber nicht unterhalb 512, geschoben werden. Normalerweise gibt das keinen Sinn, es sei denn, der Speicherbereich 828 bis 1019 wurde mit einem eigenen Maschinenprogramm belegt, und durch das Verschieben des Kassettenpuffers in höhere Regionen möchte man das Maschinenprogramm vor der Zerstörung durch ungeplante Datasetten-Operationen schützen.

Die Kenntnis der Inhalte der Speicherzellen des Kassettenpuffers kann man ausnutzen, um die ärgerlichen LOAD ERROR-Probleme zu lösen. Die Methode dazu ist im Texteinschub Nr. 36 »Reparatur von LOAD ERROR« beschrieben.

Ist die Datasette nicht angeschlossen, oder wird sie nicht eingesetzt, kann der Speicherbereich des Kassettenpuffers als freier Speicher benutzt werden.

Adresse 1020 bis 1023 ($3FC bis $3FF)

Freie Speicherplätze

Auch diese 4 Byte stehen zur freien Verfügung.

Liebe Leser, wir sind am Ziel unserer Wanderung durch die Speicherlandschaft des C 64 beziehungsweise des VC 20 angelangt.

Einzelne Beschreibungen der folgenden Speicherzellen besonders der verschiedenen Register, sind ja schon im 64'er veröffentlicht worden.

Detaillierte »Wandervorschläge« müßte ich erst zusammenstellen und ausprobieren. Deshalb mache ich bei 1023 Schluß. Ich hoffe, dieser Kurs bietet Ihnen ein kleines Nachschlagewerk und regt Sie mit den Texteinschüben und Kochrezepten zum Experimentieren an. Gerade das ist die beste Methode, Ihren Computer besser kennenzulernen.

(Dr. H. Hauck/ah)

Texteinschub Nr. 35
Dem Computer ins Wort fallen

Jedesmal, wenn ein Computer eingeschaltet wird, würden seine vielen Schaltkreise und Speicherzellen irgendwelche ungeordneten Zahlen enthalten, wenn nicht ein bestimmter Schaltkreis ein RESET-Signal erzeugte. Dieses spezielle Signal geht an alle wichtigen Teile des Computers, nämlich an den Mikroprozessor und an die Bausteine für Ein- und Ausgabe.

Dadurch wird der Computer in einen definierten Anfangszustand versetzt, in dem entweder das Betriebssystem oder, falls vorhanden, ein selbststartendes Steckmodul die Befehlsgewalt erhält.

Die fest vorgegebenen Programmschritte dieser beiden lassen jedoch ein Arbeiten mit dem Computer ohne weiteres nicht zu. Wir könnten nämlich kein Resultat an ein Ausgabegerät (Drucker, Floppy, Datasette, Bildschirm) geben, und wir könnten auch keine Daten eingeben (Tastatur, Floppy, Datasette).

Der Computer wäre nicht steuerbar, wenn wir ihn nicht in seinem vorgegebenen Programmablauf unterbrechen könnten.

Die Unterbrechungsmöglichkeit heißt in der Fachsprache »INTERRUPT«.

Im Gegensatz zu den Großrechenanlagen, die meistens mit vielen Klassen von Interrupts ausgerüstet sind, haben die Heim-Computer von Commodore nur zwei Arten:

- IRQ - der Interrupt Request
- NMI - der Non Maskable Interrupt

Ich habe nicht vor, Ihnen alle Details der Interrupt-Technik zu erklären. Das geht weit über den normalen Umfang meiner Texteinschübe hinaus. In anderen Aufsätzen können Sie mehr darüber erfahren, zum Beispiel von Helmut Welke in Ausgabe 11/84, Seite 84, oder im Assembler-Kurs von Heimo Ponnath in den Ausgaben 7 bis 9/85 und Sonderheft 8/85 (Assembler).

Aber einige Erklärungen, so hoffe ich jedenfalls, werden Ihnen auch hier das Interrupt-Prinzip deutlich machen.

Die beiden oben genannten Unterbrechungsarten unterscheiden sich sowohl dadurch, wer die Unterbrechung auslösen kann, als auch in der Art, wie sie gehandhabt werden.

NMI-Auslöser
sind Signale der RS232-Schnittstelle und der Autostart-Steckmodule. Dazu kommen noch die RESTORE-Taste, wenn sie gleichzeitig mit der RUN/STOP-Taste gedrückt wird und der CIA #2 beziehungsweise der VIA #1.

Wie gesagt, nähere Einzelheiten darüber finden Sie in den oben genannten Aufsätzen.

IRQ-Auslöser
ist 60mal in der Sekunde das Betriebssystem selbst, um die Werte von TI und TI$ höher zu setzen, um zu prüfen, ob die STOP-Taste gedrückt ist, um das Cursorblinken zu erzeugen, um die Tasten der Datasette und schließlich auch die Tastatur abzufragen. Ein IRQ-Interrupt kann aber auch durch Lesen oder Schreiben vom - beziehungsweise auf das - Band, durch die serielle Schnittstelle und durch die Rasterzeilen-Abtastung ausgelöst werden. Programmierbare IRQ-Interrupts sind möglich durch Sprite-Kollisionen, durch Lichtgriffel-Signale und durch den CIA #1 beziehungsweise den VIA #2. Besonders durch die letzteren Ein-/Ausgabe-Bausteine unterscheiden sich die Interrupts von C 64 und VC 20.

NMI-Abläufe
sind schon durch ihren Namen gekennzeichnet. »Non-Maskable« heißt soviel wie »nicht unterdrückbar«. Immer, wenn ein NMI-Signal ankommt, merkt sich der Computer, was er gerade macht, unterbindet alle IRQ-Signale und springt auf eine NMI-Routine, deren Beginn mit dem Vektor in Speicherzelle 792 und 793 vorgegeben ist.

Herr Ponnath hat im Assembler-Kurs dies sehr treffend mit dem überkochenden Kessel auf dem Herd verglichen, der heruntergestellt werden muß, selbst wenn gerade die Türglocke klingelt, was uns normalerweise beim Lesen der Zeitung unterbrechen würde.

Erst in der NMI-Routine werden nach einer vorgegebenen Prioritätsliste alle NMI-Auslöser der Reihe nach abgefragt, bis der Verursacher gefunden ist.

IRQ-Abläufe
sind maskierbar, das heißt sie können, wie gerade gesagt, unterdrückt werden, entweder durch programmiertes Abschalten - das entspricht dem Abstellen der Türglocke - oder durch ein NMI-Signal.

Bei einem IRQ-Signal wird zuerst der gerade laufende Befehl noch bearbeitet, dann startet die IRQ-Routine, deren Beginn durch den Vektor in Speicherzelle 788 und 789 vorgegeben ist. In dieser Routine wird entschieden, ob der IRQ-Interrupt durch den Maschinencode-Befehl BRK (Break) oder durch angeschlossene Peripheriegeräte ausgelöst worden ist.

Wir sehen also, daß die Unterbrechungen einer festgelegten Priorität unterworfen sind. Ihre Steuerung aber erfolgt immer so, daß keine Interrupt-Anmeldung verlorengeht, sondern jede in der gebührenden Reihenfolge abgearbeitet wird.

Schließlich sei noch hervorgehoben, daß der Sprung in die Interrupt-Routinen über die Vektoren die Möglichkeit eröffnet, diese Routinen nach eigenem Geschmack abzuändern beziehungsweise durch eigene Routinen zu ersetzen.

Texteinschub Nr. 36
Reparatur eines LOAD ERRORs

Die Datasette - das Bandgerät von Commodore - ist sicher eines der sichersten und zuverlässigsten seiner Art.

Und doch weigert sie sich gelegentlich, ein Programm vom Band in den Computer zu laden. Alles, was der Computerfreund erhält, ist die Fehlermeldung LOAD ERROR auf dem Bildschirm.

Natürlich: die nächstliegende Maßnahme ist, den LOAD-Vorgang zu wiederholen. Bringt auch das keinen Erfolg, muß die Flinte noch lange nicht ins Korn geworfen werden. Eine kleine Diagnose und die Kenntnis des Tape Headers im Kassettenspeicher (Speicherzelle 828 bis 1023) kann in den meisten Fällen weiterhelfen.

Die 1. Diagnose:

Wenn ein Programm auf Band geSAVEt wird, tut das der C 64 und VC 20 zur Sicherheit gleich zweimal, mit zwei völlig identischen Blöcken. Beim Laden des Programms wird der erste Block in den Arbeitsspeicher des Computers geladen.

Anschließend wird Byte für Byte der zweite Block vom Band mit dem ersten Block im Speicher verglichen. Übersteigt die Anzahl der dabei gefundenen Fehler ein bestimmtes Maß, dann bricht der Computer mit LOAD ERROR ab.

!! Der erste Programmblock steht aber immer noch im Arbeitsspeicher !!

Um zu sehen, ob er in Ordnung oder halbwegs brauchbar ist, machen Sie bitte nach der Fehlermeldung gar nichts - kein RUN, kein RESTORE - sondern LISTen Sie lediglich das Programm. Besteht es nur aus verfälschten Zeilen und Symbolen, dann ist nicht mehr viel zu retten.

Ist es aber fast oder völlig intakt, können wir es retten. Doch auch jetzt ist noch Vorsicht geboten. Lassen Sie das Programm in Ruhe und heben Sie sich die Korrekturen etwaiger Fehler für später auf.

Die 2. Diagnose:

Sie betrifft den Tape Header. Vor dem Laden des ersten Programmblocks in den Arbeitsspeicher kommt der Tape Header in den Kassettenpuffer (siehe den Texteinschub Nr. 20 »Tape Header«).

In Speicherzelle 828 steht ein Kennzeichen-Byte, in 829 und 830 in Low-/High-Byte-Darstellung die Adresse, ab der das Programm im Arbeitsspeicher steht.

Für uns ist aber die Adresse wichtig, die in Speicherzelle 831 und 832 steht. Sie nennt dem Betriebssystem nämlich die Endadresse des Programms im Arbeitsspeicher. Diese Adresse wird nach dem erfolgreichen Abschluß des Ladevorgangs in die Speicherzellen 45 und 46, 47 und 48, 49 und 50 eingeschrieben.

Ich sagte: »nach dem erfolgreichen Ladevorgang«. Und das gerade ist ja leider nicht eingetreten – deswegen können wir den akzeptablen ersten Programmblock im Arbeitsspeicher nicht RUNen, korrigieren und sonstwie verarzten, nur LISTen.

Reparatur:

Da durch den Abbruch die Zeiger in oben genannten drei Speicherzellenpaaren nicht gesetzt worden sind, holen wir das ganz einfach manuell nach mit der folgenden Direkteingabe:

```
POKE 45,PEEK(831): POKE 46,PEEK(832):
POKE 47,PEEK(831): POKE 48,PEEK(832):
POKE 49,PEEK(831): POKE 50,PEEK(832):
```

Das geht auch etwas eleganter und kürzer:

```
FOR K=45 TO 49 STEP 2: POKE K,PEEK(831): POKE K+1,PEEK(832):
NEXT
```

Damit sind die Zeiger richtig gesetzt, und Sie haben Ihr Programm wieder. Erst jetzt dürfen Sie eventuelle Fehler korrigieren.

Ich habe nicht erwähnt, was die Zeiger in 45 und 46, 47 und 48, 49 und 50 bedeuten. Aber das steht ja schließlich in der Memory Map.

```
6000 FOR A=828 TO 853
6010 READ B
6020 POKE A,B
6030 NEXT:END
6040 DATA 120,169,73,141,20,3,169,3,
6050 DATA 141,21,3,88,96,169,1,141,139
6060 DATA 2,169,0,141,140,2,76,49,234
```

Listing 1. Programm zur Änderung der Tastenwiederholgeschwindigkeit

```
828  SEI          setzt die Interrupt Enable Flagge
829  LDA #73      lädt Akku mit der Zahl 73
831  STA 788      schreibt die 73 in Zelle 788
834  LDA #3       lädt Akku mit der Zahl 3
836  STA 789      schreibt die 3 in die Zelle 789
839  CLI          löscht die Interrupt Enable Flagge
840  RTS          Ende des Unterprogramms
841  LDA #1       lädt Akku mit der Zahl 1
843  STA 651      schreibt die 1 in Zelle 651
846  LDA #0       lädt Akku mit der 0
848  STA 652      schreibt die 0 in die Zelle 652
851  JMP 59953    Sprung auf Speicherzelle 59953 zum  Weiterlauf
der normalen Interrupt-Routine
```

Listing 2. Disassembliertes Programm

```
,033C  78        SEI
,033D  A9 49     LDA #49
,033F  8D 14 03  STA 0314
,0342  A9 03     LDA #03
,0344  8D 15 03  STA 0315
,0347  58        CLI
,0348  60        RTS
,0349  A9 01     LDA #01
,034B  8D 8B 02  STA 028B
,034E  A9 00     LDA #00
,0350  8D 8C 02  STA 028C
,0353  4C 31 EA  JMP EA31
```

Listing 3. Disassembliertes Programm mit Hexdump

Inhaltsverzeichnis der Texteinschübe und Tabellen

Texteinschub Nr. 1:	Der USR-Befehl (VC 20)
Texteinschub Nr. 2.	Die Low-/High-Byte-Darstellung
Texteinschub Nr. 3:	Manipuliertes Basic (C64)
Texteinschub Nr. 4:	Zeiger, Vektoren und Flaggen
Texteinschub Nr. 5:	Die Zahlendarstellung bei den Commodore-Computern
Texteinschub Nr. 6:	Was ist ein Stapelspeicher (Stack)?
Texteinschub Nr. 7:	Der sichtbare Basic-Speicher
Texteinschub Nr. 8:	Normale Variable in Basic
Texteinschub Nr. 9:	Darstellung der normalen Variablen im Speicher
Texteinschub Nr. 10:	Felder in Basic
Texteinschub Nr.11:	Darstellung der Felder-Variablen im Speicher
Texteinschub Nr. 12:	Darstellung der Variablen einer selbstdefinierten Funktion
Texteinschub Nr. 13:	Wie zufällig sind Zufallszahlen?
Texteinschub Nr. 14:	ST-atus
Texteinschub Nr. 15:	Dynamische Tastenabfrage
Texteinschub Nr. 16:	Die eingebaute Uhr
Texteinschub Nr. 17:	Experimente mit dem Kassetten-Puffer
Texteinschub Nr. 18:	Fehlererkennung mit Parity-Bits
Texteinschub Nr. 19:	Files - Geräte - Namen - Nummern
Texteinschub Nr. 20:	Tape-Header
Texteinschub Nr. 21:	Abfrage der Tasten-Codes oder 476 Funktionstasten
Texteinschub Nr. 22:	Cursor-Spiele oder der INPUT-Befehl einmal etwas anders
Texteinschub Nr. 23:	Logische und echte Zeilen
Texteinschub Nr. 24:	Der Stapelspeicher
Texteinschub Nr. 25:	Programme, die sich selbst verändern
Texteinschub Nr. 26:	Bunte Zeichen und bunter Hintergrund
Texteinschub Nr. 27:	Turbo-Tasten
Texteinschub Nr. 28:	Schnittstelle und Port
Texteinschub Nr. 29:	Die Elemente der RS232-Schnittstelle
Texteinschub Nr. 30:	Die Programmierung der RS232-Schnittstelle
Texteinschub Nr. 31:	Indirekte Sprung-Vektoren
Texteinschub Nr. 32:	Die Kurzschrift von Basic
Texteinschub Nr. 33:	Der vorbereitete SYS-Befehl
Texteinschub Nr. 34:	Das Mauerblümchen USR (C 64)
Texteinschub Nr. 35:	Dem Computer ins Wort fallen
Texteinschub Nr. 36:	Reparatur eines LOAD ERRORs
Anhang:	Die Bedeutung der Speicherzahlen 0 bis 1023, nach Funktionen geordnet.
Tabelle 1:	Unterschiede im Speicher zwischen VC 20 und C 64
Tabelle 2:	RAM/ROM-Umschaltung beim C64:
Tabelle 3:	Beginn des Programmspeichers
Tabelle 4:	Ende des Programmspeichers
Tabelle 5:	Liste der Gerätenummern
Tabelle 6:	Funktionen der Sekundäradressen
Tabelle 7:	Tape-Header Kennzahlen
Tabelle 8:	Tape-Header Bytes
Tabelle 9:	Alle Tasten-Codes
Tabelle 10:	Farben und ihre Codes beziehungsweise Tasten
Tabelle 11:	Die Bedeutung der Bits im RS232-Steuerregister
Tabelle 12:	Das RS232-Befehlsregister
Tabelle 13:	Realisierbarkeit der Schnittstellen am C64 und VC 20
Tabelle 14:	Das RS232-Statusregister
Tabelle 15:	Flagge für den RS232-Interrupt

Die Bedeutung der Speicherzellen 0 bis 1023, nach Funktionen geordnet

Bandoperationen

146	$92	Zeitkonstante beim Lesen vom Band
147	$93	Flagge für LOAD oder VERIFY
150	$96	Arbeitsspeicher für Band-Leseroutinen
153	$99	Nummer des Eingabegerätes
155	$9B	Fehlerkontrolle bei Bandoperationen
156	$9C	Flagge für korrektes Byte vom Band
158-159	$9E-$9F	Zwischenspeicher bei Kassettenoperationen

165	$A5	Zähler für Band-Synchronisierung
167	$A7	Zwischenspeicher für Kassettenroutinen
168	$A8	Bitzähler bei Band-Ein-/Ausgabe
170	$AA	Zwischenspeicher für Kassettenroutinen
171	$AB	Quersummenprüfung und Zähler für Band-Header
172-173	$AC-$AD	Zeiger auf die Anfangsadresse für Ein-/Ausgabe
174-175	$AE-$AF	Zeiger auf die Endadresse für Ein-/Ausgabe
176-177	$B0-$B1	Zeitkonstante beim Lesen vom Band
178-179	$B2-$B3	Zeiger auf den Kassettenpuffer
181	$B5	Blockangabe bei Kassettenoperationen
182	$B6	Ausgabe-Zwischenspeicher
183	$B7	Länge des File-Namens
185	$B9	Sekundär-Adresse
186	$BA	Geräte-Nummer
187-188	$BB-$BC	Zeiger auf Adresse des derzeitigen File-Namens
189	$BD	Zwischenspeicher für Zeichen
190	$BE	Blockzähler für Kassetten-Ein-/Ausgabe
191	$BF	Zwischenspeicher für LOAD-Operationen vom Band
192	$CO	Sperre des Motors der Datasette
193-194	$C1-$C2	Anfangsadresse für Ein-/Ausgabe-Operationen
195-196	$C3-$C4	Zeiger auf den Anfang des Programms hinter dem Tape Header
256-318	$100-$13E	Arbeitsspeicher für Fehler bei der Eingabe vom Band
Bildschirm-Cursor		
9	$9	Spaltenposition des Cursors vor dem letzten TAB- oder SPC-Befehl
200	$C8	Zeiger auf das Ende der eingegebenen logischen Zeile
201-202	$C9-$CA	Zeiger auf Zeilen- und Spaltenposition des letzten Zeichens einer Zeile
204	$CC	Schalter für Cursorblinken
205	$CD	Zähler für Blinkfrequenz des Cursors
206	$CE	Bildschirmcode des Zeichens unter dem Cursor
207	$CF	Flagge für Blinkzustand des Cursors
209-210	$D1-$D2	Zeiger auf den Anfang der Bildschirmzeile, auf welcher der Cursor gerade steht
211	$D3	Position des Cursors innerhalb einer logischen Zeile
214	$D6	Nummer der echten Zeile, in der sich der Cursor gerade befindet
647	$287	Zeichenfarbe unter dem Cursor
Bildschirm-Farbe		
243-244	$F3-$F4	Position des Cursors im Farbspeicher
646	$286	Aktuelle Farbe der Zeichen (Vordergrundfarbe)
647	$287	Zeichenfarbe unter dem Cursor
Bildschirm-Zeichen		
199	$C7	Flagge für reverse Darstellung der Zeichen
206	$CE	Bildschirmcode des Zeichens unter dem Cursor
212	$D4	Flagge für Gänsefuß-Modus
215	$D7	Zwischenspeicher für den ASCII-Codewert der zuletzt gedrückten Taste
216	$D8	Flagge für Insert-Modus
Bildschirm-Zeilen		
200	$C8	Zeiger auf das Ende der eingegebenen logischen Zeile
201-202	$C9-$CA	Zeiger auf Zeilen- und Spaltenposition des letzten Zeichens einer Zeile
209-210	$D1-$D2	Zeiger auf den Anfang der Bildschirmzeile, auf welcher der Cursor gerade steht
211	$D3	Position des Cursors innerhalb einer logischen Zeile
213	$D5	Länge einer Bildschirmzeile
214	$D6	Nummer der echten Zeile, auf der sich der Cursor gerade befindet
217-242	$D9-$F2	Link-Tabellen der Bildschirm-Zeilen
658	$292	Flagge für Scrollen
DATA		(siehe READ)
Datei		
152	$98	Anzahl der offenen Dateien
153	$99	Nummer des Eingabegerätes
154	$9A	Nummer des Ausgabegerätes
183	$B7	Länge des derzeitigen Datei-Namens
184	$B8	Nummer der derzeitigen Datei
185	$B9	Derzeitige Sekundär-Adresse
186	$BA	Derzeitige Geräte-Nummer
187-188	$BB-$BC	Zeiger auf Adresse des derzeitigen Datei-Namens
601-610	$259-$262	Tabelle der Datei-Nummern
611-620	$263-$26C	Tabelle der Geräte-Nummern
621-630	$26D-$276	Tabelle der Sekundär-Adressen
DIM		
11	$B	Anzahl der Dimensionen von Feldern (Arrays)
12	$C	Flagge für Basic-Routinen, die ein Feld suchen, beziehungsweise aufbauen
Eingabe-Puffer		
7	$7	Suchzeichen zur Prüfung von Basic-Texteingabe
8	$8	Suchzeichen speziell für Befehlsende und Gänsefüße
11	$B	Flagge für den Eingabe-Puffer
512-600	$200-$258	Eingabe-Puffer von Basic
Einschalten/Reset (beeinflußte Adressen)		
0-2	$0-$2	Sprungbefehl und wählbare Sprungadresse beim USR-Befehl (nur VC 20)
3-4	$3-$4	Vektor auf die Routine zur Umwandlung einer Gleitkommazahl in eine ganze Zahl mit Vorzeichen
5-6	$5-$6	Vektor auf die Routine zur Umwandlung einer ganzen Zahl in eine Gleitkommazahl
19	$13	Flagge zur Kennzeichnung des laufenden Ein-/Ausgabegerätes
22	$16	Zeiger auf freien Speicherplatz im String Descriptor Stack
43-44	$2B-$2C	Zeiger auf Anfang der Basic-Programme im Speicher
45-46	$2D-$2E	Zeiger auf Anfang der Variablen im Speicher (nur bei Reset)
51-52	$33-$34	Zeiger auf die untere Grenze des Speicherbereichs für den Text der Zeichenketten-Variablen
55-56	$37-$38	Zeiger auf das Ende des für Basic-Programme verfügbaren Speichers
122-123	$7A-$7B	Teil der CHRGET-Routine
139-143	$8B-$8F	Wert der RND-Funktion als Gleitkommazahl
153	$99	Nummer des Eingabe-Gerätes
154	$9A	Nummer des Ausgabe-Gerätes
160-162	$A0-$A2	Interne Uhr für TI und TI$ (nur beim Einschalten)
178-179	$B2-$B3	Zeiger auf den Kassetten-Puffer
195-196	$C3-$C4	Zeiger auf den Anfang des Programms hinter dem Tape Header
256-511	$100-$1FF	Stapelspeicher (Stack)
641-642	$281-$282	Zeiger auf den Anfang des Programmspeichers
643-644	$283-$284	Zeiger auf das Ende des Programmspeichers
646	$286	Aktuelle Farbe der Zeichen (Vordergrundfarbe)
648	$288	Beginn des Bildschirmspeichers
655-656	$28F-$290	Vektor auf die Routine der Tastencode-Tabellen
784-786	$310-$312	nur C64, identisch mit 0-3 beim VC 20
788-819	$314-$333	Indirekte Sprungvektoren auf Routinen des Betriebssystems
END		
57-58	$39-$3A	Nummer der laufenden Basic-Programmzeile
59-60	$3B-$3C	Zeilennummer der letzten Programmunterbrechung
61-62	$3D-$3E	Zeiger auf die Adresse, ab welcher der Text der laufenden Basic-Zeile abgespeichert ist
Felder (Arrays)		
11	$B	Anzahl der Dimensionen von Feldern (Arrays)
12	$C	Flagge für Basic-Routinen, die ein Feld suchen, beziehungsweise aufbauen
16	$10	Flagge zur Anzeige eines Variablenfeldes oder einer selbstdefinierten Funktion
47-48	$2F-$30	Zeiger auf die Anfangsadresse des Speicherbereiches für Felder (Arrays)
49-50	$31-$32	Zeiger auf die Endadresse des Speicherbereichs für Felder (Arrays)
FN		
16	$10	Flagge zur Anzeige eines Variablenfeldes oder einer selbstdefinierten Funktion
78-79	$4E-$4F	Zeiger auf Adresse, ab welcher der Wert der Variablen einer selbst definierten Funktion gespeichert ist
FOR-NEXT		
47-48	$2F-$30	Zeiger auf die Anfangsadresse des Speicherbereichs für Felder (Arrays)
57-58	$39-$3A	Nummer der laufenden Basic-Programmzeile
73-74	$49-$4A	Zwischenspeicher für Variable einer FOR-NEXT-Schleife
FRE		
49-50	$31-$32	Zeiger auf die Endadresse des Speicherbereichs für Felder (Arrays)
51-52	$33-$34	Zeiger auf die untere Grenze des Speicherbereichs für den Text der Zeichenketten-Variablen
Garbage Collection		
15	$F	Flagge bei LIST, Garbage Collection und Textumwandlung
49-50	$31-$32	Zeiger auf die Endadresse des Speicherbereichs für Felder (Arrays)
51-52	$33-$34	Zeiger auf die untere Grenze des Speicherbereichs für den Text der Zeichenketten-Variablen
83	$53	Flagge für Garbage Collection
GET		
17	$11	Flagge für INPUT, GET oder READ
18	$13	Flagge zur Kennzeichnung des laufenden Ein-/Ausgabegerätes
67-68	$43-$44	Zeiger auf die Adresse, aus welcher die Befehle INPUT, GET und READ die Zeichen/Zahlen holen
GET #		
19	$13	Flagge zur Kennzeichnung des laufenden Ein-/Ausgabegerätes
153	$99	Nummer des Eingabegerätes
Gleitkomma		
3-4	$3-$4	Vektor auf die Routine zur Umwandlung einer Gleitkommazahl in eine ganze Zahl mit Vorzeichen
5-6	$5-$6	Vektor auf die Routine zur Umwandlung einer ganzen Zahl in eine Gleitkommazahl

97-102	$61-$66	Gleitkomma-Akkumulator Nr. 1
104	$68	Überlauf-Speicher des Gleitkomma-Akkumulators Nr.1
105-110	$69-$6E	Gleitkomma-Akkumulator Nr. 2
111	$6F	Flagge für Vorzeichenvergleich der Gleitkomma-Akkumulatoren 1 und 2
112	$70	Rundungsspeicher des Gleitkomma-Akkumulators Nr.1
255	$FF	Zwischenspeicher von Gleitkommazahlen in ASCII-Werte
256-266	$100-$10A	Arbeitsspeicher für Umwandlung von Gleitkommazahlen in ASCII-Werte
778-779	$30A-$30B	Indirekter Sprungvektor auf die Basic-Routine, die einen numerischen Ausdruck in eine Gleitkommazahl umwandelt
GOTO		
20-21	$14-$15	Zeilennummer für LIST, GOTO, GOSUB und ON
57-58	$39-$3A	Nummer der laufenden Basic-Programmzeile
INPUT		
17	$11	Flagge für INPUT, GET oder READ
67-68	$43-$44	Zeiger auf die Adresse, aus welcher die Befehle INPUT, GET und READ die Zeichen/Zahlen holen
INPUT #		
19	$13	Flagge zur Kennzeichnung des laufenden Ein-/Ausgabegerätes
153	$99	Nummer des Eingabegerätes
INST		
212	$D4	Flagge für Gänsefuß-Modus
216	$D8	Flagge für INSERT-Modus
Interrupt mit BREAK		
57-58	$39-$3A	Nummer der laufenden Basic-Programmzeile
170	$AA	Zwischenspeicher für Kassettenroutinen
663	$297	RS232-Status-Register
790-791	$316-$317	Vektor auf die BREAK-Interrupt-Routine
Interrupt mit IRQ		
671-672	$29F-$2A0	Zwischenspeicher für den IRQ-Vektor während Kassetten-Ein-/Ausgabe
788-789	$314-$315	Vektor auf die IRQ-Interrupt-Routine
Interrupt mit NMI		
792-793	$318-$319	Vektor auf die NMI-Interrupt-Routine
Kassettenpuffer		
166	$A6	Zähler der bearbeiteten Bytes im Kassettenpuffer
178-179	$B2-$B3	Zeiger auf den Kassettenpuffer
828-1019	$33C-$3FB	Kassettenpuffer
LIST		
15	$F	Flagge bei LIST, Garbage Collection und Textumwandlung
20-21	$14-$15	Zeilennummer für LIST, GOTO, GOSUB und ON
LOAD/VERIFY		
10	$A	Flagge für LOAD oder VERIFY
147	$93	Flagge für LOAD oder VERIFY
172-173	$AC-$AD	Zeiger auf die Anfangsadresse für Ein-/Ausgabe
174-175	$AE-$AF	Zeiger auf die Endadresse für Ein-/Ausgabe
183	$B7	Länge des File-Namens
185	$B9	Sekundär-Adresse
187-188	$BB-$BC	Zeiger auf Adresse des derzeitigen File-Namens
195-196	$C3-$C4	Zeiger auf den Anfang des Programms hinter dem Tape Header
816-817	$330-$331	Indirekter Sprungvektor auf die LOAD-Routine des Betriebssystems
NEXT (siehe FOR)		
READ DATA		
17	$11	Flagge für INPUT, GET oder READ
63-64	$3F-$40	Zeilennummer des gerade laufenden DATA-Befehls
65-66	$41-$42	Zeiger auf die Adresse, ab der die laufenden DATA-Angaben gespeichert sind
67-68	$43-$44	Zeiger auf die Adresse, aus welcher die Befehle INPUT, GET und READ die Zeichen/Zahlen holen
75-76	$4B-$4C	Zwischenspeicher für Zeiger bei READ und bei mathematischen Operationen
RESET (siehe Einschalten)		
RND		
139-143	$8B-$8F	Wert der RND-Funktion als Gleitkommazahl
RS232-Schnittstelle		
167	$A7	Zwischenspeicher für Eingabe über die RS232-Schnittstelle
168	$A8	Bitzähler für RS232-Eingabe
169	$A9	RS232-Flagge für Startbit-Prüfung
170	$AA	RS232-Eingabespeicher
171	$AB	Parityprüfung
181	$B5	RS232-Anzeige für nächstes Bit
182	$B6	Ausgabe-Zwischenspeicher für RS232
189	$BD	Zwischenspeicher für RS232-Parity-Prüfung
247-248	$F7-$F8	Zeiger auf den Anfang des RS232-Eingabepuffers
249-250	$F9-$FA	Zeiger auf den Anfang des RS232-Ausgabepuffers
659	$293	RS232-Steuerregister
660	$294	RS232-Befehlsregister
661-662	$295-$296	RS232 frei wählbare Baudrate
663	$297	RS232-Statusregister
664	$298	RS232-Anzahl der zu übertragenden Bits
665-666	$299-$29A	Zeit, die bei RS232 zum Übertragen eines Bits gebraucht wird
667	$29B	Index auf das Ende des RS232-Eingabepuffers
668	$29C	Index auf den Anfang des RS232-Eingabepuffers
669	$29D	Index auf den Anfang des RS232-Ausgabepuffers
670	$29E	Index auf das Ende des RS232-Ausgabepuffers
SAVE		
172-173	$AC-$AD	Zeiger auf die Anfangsadresse für Ein-/Ausgabe
174-175	$AE-$AF	Zeiger auf die Endadresse für Ein-/Ausgabe
818-819	$332-$333	Indirekter Sprungvektor auf die SAVE-Routine des Betriebssystems
Serielle Schnittstelle		
148	$94	Flagge für Floppy/Drucker-Ausgabe
149	$94	Zeichen im Ausgabepuffer
163-164	$A3-$A4	Zwischenspeicher
172-173	$AC-$AD	Zeiger auf die Anfangsadresse für Ein-/Ausgabe
174-175	$AE-$AF	Zeiger auf die Endadresse für Ein-/Ausgabe
193-194	$C1-$C2	Anfangsadresse für Ein-/Ausgabe-Operationen
SIN		
18	$12	Flagge für Vorzeichen des Ergebnisses bei SIN und TAN
Speicherbelegung		
43-44	$2B-$2C	Zeiger auf den Anfang der Basic-Programme im Speicher
45-46	$2D-$2E	Zeiger auf die Anfangsadresse des Speicherbereichs für Variable
47-48	$2F-$30	Zeiger auf die Anfangsadresse des Speicherbereichs für Felder (Arrays)
49-50	$31-$32	Zeiger auf die Endadresse +1 des Speicherbereichs für Felder (Arrays)
51-52	$33-$34	Zeiger auf die untere Grenze des Speicherbereichs für den Text der String-Variablen
53-54	$35-$36	Zeiger auf die Adresse des zuletzt eingegebenen Strings
55-56	$37-$38	Zeiger auf das Ende des für Basic-Programme verfügbaren Speichers
641-642	$281-$282	Zeiger auf den Anfang des Programmspeichers
643-644	$283-$284	Zeiger auf das Ende des Programmspeichers
648	$288	Beginn des Bildschirmspeichers
Speicher zur freien Verfügung		
146-150	$92-$96	nur wenn Datasette nicht benutzt wird
163-177	$A3-$B1	nur wenn Datasette oder RS232-Schnittstelle nicht benutzt wird
247-250	$F7-$FA	nur wenn RS232-Schnittstelle nicht benutzt wird
251-254	$FB-$FE	
659-670	$293-$29E	nur wenn RS232-Schnittstelle nicht benutzt wird
671-672	$29F-$2A0	nur wenn Datasette nicht benutzt wird
673-678	$2A1-$2A6	nur beim VC 20
679-767	$2A7-$2FF	
784-787	$310-$313	nur beim VC 20
820-827	$334-$33B	
828-1019	$33C-$3FB	nur wenn Datasette nicht benutzt wird
1020-1023	$3FC-$3FF	
ST(atus)		
144	$90	Status-Variable ST
663	$297	RS232-Statusregister
Stapelspeicher (Stack)		
25-33	$19-$21	Stack für vorläufige Zeichenketten
319-511	$13F-$1FF	Speicherbereich des Mikroprozessor-Stapels
STOP		
57-58	$39-$3A	Nummer der laufenden Basic-Programmzeile
59-60	$3B-$3C	Zeilennummer der letzten Programmunterbrechung
145	$91	Zwischenspeicher für Abfrage der STOP-Taste
808-809	$328-$329	Indirekter Sprungvektor auf die STOP-Routine des Betriebssystems
STRINGS		
22	$16	Zeiger auf freien Speicherplatz im String Descriptor Stack
23-24	$17-$18	Zeiger auf die Adresse der letzten Zeichenkette im Temporary String Stack
25-33	$19-$21	Descriptor Stack für vorläufige Zeichenketten
51-52	$33-$34	Zeiger auf die untere Grenze des Speicherbereichs für den Text der String-Variablen
53-54	$35-$36	Zeiger auf die Adresse des zuletzt eingegebenen Strings
80-82	$50-$52	Zeiger auf einen vorläufigen Speicherplatz einer Zeichenkette, die gerade bearbeitet wird
SYS		
780	$30C	Speicher für den Akkumulator
781	$30D	Speicher für das X-Register
782	$30E	Speicher für das Y-Register
783	$30F	Speicher für das Status-(P)-Register

Fortsetzung von S. 65

MEMORY MAP mit Wandervorschlägen

TAN		
18	$12	Flagge für Vorzeichen des Ergebnisses bei SIN und TAN
Tastatur		
145	$91	Zwischenspeicher für Abfrage der STOP-Taste
197	$C5	Tasten-Code der zuletzt gedrückten Taste
198	$C6	Anzahl der Zeichen im Tastaturpuffer
203	$CB	Tasten-Code der gerade gedrückten Taste
208	$D0	Flagge für Eingabe von Tastatur oder Bildschirm
245-246	$F5-$F6	Vektor auf die Decodiertabelle für ASCII-Codewerte der Tasten
631-640	$277-$280	Tastaturpuffer
649	$289	Maximale Länge des Tastaturpuffers
650	$28A	Flagge für Tastenwiederholung
651	$28B	Zähler für Wiederholungsgeschwindigkeit der Tasten
652	$28C	Zähler für Ansprechzeit der Wiederholfunktion von Tasten
653	$28D	Tastencode der SHIFT-, CTRL- und Commodore-Taste
654	$28E	Tastencode der zuletzt gedrückten SHIFT-, CTRL- und Commodore-Taste
655-656	$28F-$290	Vektor auf die Routine der Tastencode-Tabellen
657	$291	Flagge für Verriegelung der Zeichensatz-Umschaltung
Token		
8	$8	Suchzeichen speziell für Befehlsende und Gänsefüße
11	$B	Anzahl der Dimensionen von Feldern (Arrays)
15	$F	Flagge bei LIST, Garbage Collection und Text-umwandlung
61-62	$3D-$3E	Zeiger auf die Adresse, ab welcher der Text der laufenden Basic-Zeile abgespeichert ist
122-123	$7A-$7B	Teile der CHRGET-Routine
512-600	$200-$258	Basic-Eingabepuffer
772-773	$304-$305	Indirekter Sprungvektor auf die Basic-Routine, die ASCII-Text in Tokens umwandelt
774-775	$306-$307	Indirekter Sprungvektor auf die Basic-Routine, die Tokens in ASCII-Text zurückwandelt (List)
776-777	$308-$309	Indirekter Sprungvektor auf die Basic-Routine, die den nächsten Befehl liest und ausführt
Uhr		
160-162	$A0-$A2	Interne Uhr für TI und TI$
USR		
784-786	$310-$312	Nur C64; Sprungbefehl und wählbare Sprungadresse des USR-Befehls
0-2	$0-$2	Nur VC 20: Sprungbefehl und wählbare Sprungadresse des USR-Befehls
Variable		
13	$D	Flagge zur Bestimmung des Variablentyps (String oder Zahl)
14	$E	Flagge zur Bestimmung des Variablentyps (ganze Zahl oder Gleitkommazahl)
45-46	$2D-$2E	Zeiger auf die Anfangsadresse des Speicherbereichs für Variable
47-48	$2F-$30	Zeiger auf die Anfangsadresse des Speicherbereichs für Felder (Arrays)
49-50	$31-$32	Zeiger auf die Endadresse +1 des Speicherbereichs für Felder (Arrays)
51-52	$33-$34	Zeiger auf die untere Grenze des Speicherbereichs für den Text der Stringvariablen
53-54	$35-$36	Zeiger auf die Adresse des zuletzt eingegebenen Strings
69-70	$45-$46	Name der gerade aufgerufenen Basic-Variablen
71-72	$47-$48	Zeiger auf die Adresse des Wertes der gerade aufgerufenen Basic-Variablen
73-74	$49-$4A	Zwischenspeicher für Variable einer FOR-NEXT-Schleife für diverse Basic-Befehle
Vektoren für indirekte Sprünge		
768-779	$300-$30B	Vektoren auf Routinen des Basic-Übersetzers (Interpreters)
794-819	$31A-$333	Vektoren auf Routinen des Betriebssystems (Kernel)

Achtung C-Programmierer aufgepaßt!

Jetzt gibt es Small-C, ein komplettes Entwicklungssystem im CP/M-Modus für den Commodore 128 PC. Mit Editor, Compiler, Linker und vielen weiteren Utilities.

Alle Programme sind in Small-C geschrieben, der Quellcode wird mitgeliefert. So können Sie das Entwicklungssystem nach eigenen Wünschen und Erfordernissen erweitern und modifizieren.

Das Programmpaket enthält:

- **Small-C-Compiler**
- **Small-Mac: Assembler und Utilities**
- **Small-Tools: Editor und Text-Tools**

Hardware-Anforderungen:
C128/C128 D, Diskettenlaufwerk 1571.

Bestell-Nr. MS 483 (5¼"-Diskette)

Für nur DM 148.-* (sFr. 132.-/öS 1490,-*)

*inkl. MwSt., unverbindliche Preisempfehlung.

Wenn Sie direkt beim Verlag bestellen wollen: Gegen Vorauskasse durch Verrechnungsscheck oder mit der abgedruckten Zahlkarte.

Bestellungen im Ausland bitte an untenstehende Adressen:
Schweiz: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug
Österreich: Ueberreuter Media Verlagsges. mbH, Alser Straße 24, A-1091 Wien

Markt&Technik
Unternehmensbereich Buchverlag
Hans-Pinsel-Straße 2, 8013 Haar bei München

Markt&Technik
128er-Software

Übrigens:
Small-C gibt's auch für die Schneider-Computer.
Zum gleichen Preis!
Best.-Nr. MS 484

Dr. Dobb's Journal
J. E. Hendrix

Small-C
Entwicklungssystem
C-Compiler

8080-/Z80-Makro-Assembler · Linker/Loader
Bibliotheksverwalter · Editor/Text-Tools

Für Commodore 128 (128 D)
Floppy 1571-Format

Alle Programme mit Quellcode!