Memory Map mit Wandervorschlägen

TEIL 1

Es steckt sehr viel im ersten Kilobyte des VC 20 und C 64. Wir werden Ihnen im Rahmen dieses Kurses die Bedeutung und Anwendung der Speicher und Register von Betriebssystem und Interpreter näherbringen.

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 Home-Computer 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 und Anwendungen der PEEK- und POKEbaren Adressen, — sozusagen eine Wanderkarte mit Tourenvorschlägen und Sehenswürdigkeiten — in Form von Beispielen und Kochrezepten, näher zu bringen. 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«.

■ Zum Adressenbereich:
Prinzipiell sind natürlich alle RAM-Adressen (RAM = Lese- und Schreibspeicher) POKEbar und kämen daher in Betracht. Vorerst aber werden wir uns 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 auf Seite 137.

Tabelle 1.
Unterschiede zwischen VC 20 und C 64

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

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 Ent-

Memory Map

scheidungsfähigkeit, 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 — $2) beim VC 20: Sprungbefehl und wählbare »Sprungadresse« des USR-Befehls.

Die drei Adressen werden bei der Abwicklung des Basic-Befehls USR verwendet und stehen dem Programmierer zur Verfügung.

Hinweise: 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.

Hand aufs Herz: Haben Sie USR schon einmal benützt? 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.

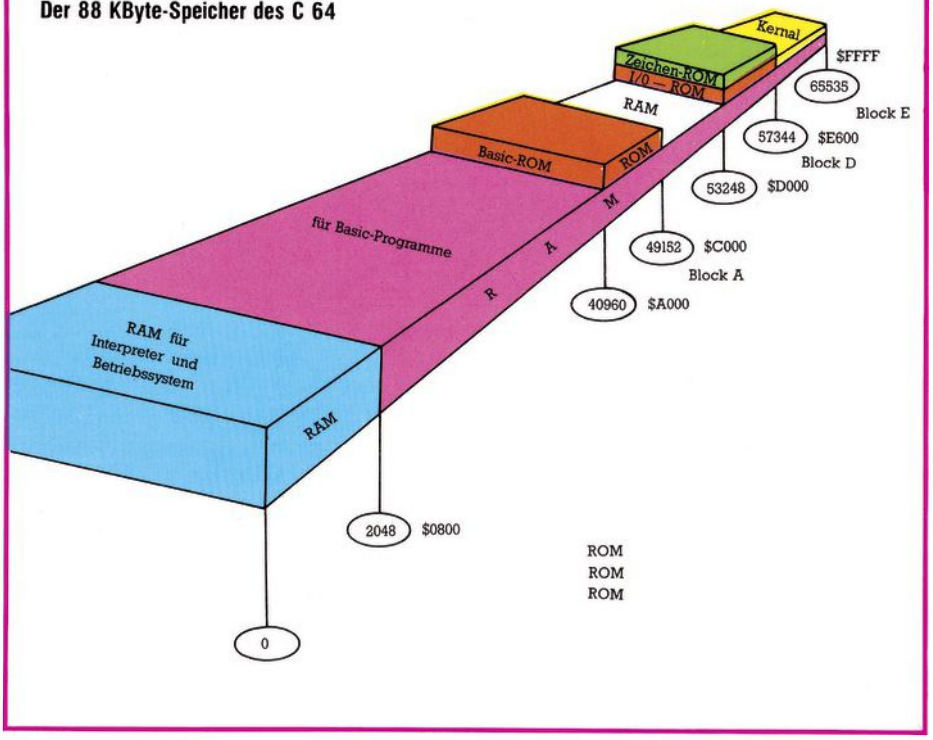
Beispiel — Sprung auf 56524 ($DCCC):

mit SYS: SYS 56524

mit USR: POKE 1,204:POKE 2,220:X = USR(Y)

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

Der 88 KByte-Speicher des C 64



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 gan-

mit Wandervorschlägen

zen 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 selbtgeschriebenen 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 GOSUB. 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.

Jetzt aber wollen wir uns anschauen, wie diese drei Speicherzellen beim C 64 verwendet werden.

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.

Datenrichtungs-Register 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 Bits (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 — 49151 ($A000 — $BFFF) zwischen dem Basic-Übersetzer (Interpreter) im ROM und freiem RAM um (Normalzustand = 1)

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

Bit 2
schaltet den Speicherbereich 53248 — 57343 ($D000 — $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)

Als erstes möchte ich die RAM-ROM-Umschaltung näher beschreiben.

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 Bild 1) nur 64 KByte adressierbar sind, müssen die restlichen 22KByte bei Bedarf eingeschoben werden — und das machen die oben erwähnten Bits 0 — 2 des Datenregisters.

In Bild 2 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:
— 40960-49151 ($A000-$BFFF) = BLOCK A
— 53248-57343 ($D000-$DFFF) = BLOCK D
— 57344-65535 ($E000-$FFFF) = BLOCK E

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

Memory Map

# 2 1 0	BLOCK A	BLOCK D	BLOCK E
1 1 1	Basic	I/0	Kernal
1 1 0	RAM	I/0	Kernal
1 0 1	RAM	I/0	RAM
1 0 0	RAM	RAM	RAM
0 1 1	Basic	Zeichen	Kernal
0 1 0	RAM	Zeichen	Kernal
0 0 1	RAM	Zeichen	RAM
0 0 0	RAM	RAM	RAM

Dabei bedeuten
— Basic: Basic-Übersetzer (Interpreter)
— I/0: 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.

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 oben genannten »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 den RAM ein mit:

```
POKE 1,54
```

Wir merken natürlich noch keinen Unterschied, denn das RAM-Basic ist ja noch dasselbe, wie es im ROM steht.

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 in 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 (siehe Literatur), 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 POKE1,53 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. Im Grafik-Kurs war das ausführlich beschrieben.

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 benützt werden. Eine genaue Beschreibung der dadurch erzeugten sinnvollen Speicherkombinationen finden Sie in dem Buch »64 Intern« von Data Becker ab Seite 14.

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

mit Wandervorschlägen

Motor-Tasten der Datasette (PLAY, REWIND, FAST FORWARD) gedrückt ist.
Zur Probe:

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

Betrieb des Kassettenrecorders

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 gleich nachher 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 GEDRÜCKT«
```

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 Datenrichtungsregister (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 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. Das nächste Mal wird unsere Speicherreise weitergehen, wobei natürlich nicht alle Speicherzellen soviel hergeben beziehungsweise dem Programmierer so direkt zur Verfügung stehen, wie die Adressen 0 und 1.

(Dr. Helmuth Hauck/aa)

Literatur:
1) H. Ponnath, »Reise durch das Wunderland der Grafik«, 64'er, Ausgabe 4/84 und folgende
2) J.Butterfield, »The Architecture of the 64«, Commodore User, Sept. 1983
3) M.Angerhausen et al., VC 20 Intern, Data Becker, 1983
4) M.Angerhausen et al., 64 Intern, S. 11-21 Data Becker, 1983
5) S.Leemon, Mapping the C 64, COMPUTE Publications, 1984

Die Low-High-Byte-Methode

DARSTELLUNG VON ADRESSEN GRÖSSER 255

Um große Zahlen darzustellen, wird von allen Homecomputern die sogenannte High-Low-Byte-Methode angewendet.

Eine Speicherzelle der Commodore Computer ist 8 Bit lang, das ist 1 Byte, und sie kann daher als größte Zahl 255 ($FF) enthalten. Für Zahlen größer als 255 hängen wir mehrere Speicherzellen hintereinander, in unserem Fall deren zwei. Mit 2 Bytes (16 Bit) können wir nämlich maximal 65535 ($FFFF) darstellen. Diese Aufspaltung einer Adresse in zwei Speicherzellen soll das folgende Beispiel verdeutlichen.

dezimal	**47491**			
dual	1011	1001	1000	0011
hex $	B	9	8	3
High-Byte	185			
Low-Byte			131	

Sie sehen, daß die Dezimalzahl 47491 aufgespalten wird in $B9 = 185 und $83 = 131. Zur Erinnerung: Jede Stelle einer Hex-Zahl kann direkt in eine 4stellige Dualzahl und umgekehrt gewandelt werden (1011 = $B, 1000 = $8).

Der Umrechnungsweg über eine 16stellige Dualzahl ist natürlich viel zu aufwendig. Ich empfehle Ihnen folgendes Kochrezept:

(1) Dezimal High/Low-Byte:

```
47491 : 256 = 185, Rest 131
```

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

```
185 * 256 — 47491 = -131
```

2) High/Low-Byte Dezimal:

```
HB* 256 + LB = Dezimal
185* 256 + 131 = 47491
```

Wichtige Regel:
Die Mikroprozessoren von VC 20 und C 64 verlangen, daß immer das Low-Byte vor dem High-Byte kommen muß. Die Zahl wird sozusagen von rechts nach links gelesen (im Beispiel: 131, 185)