Memory Map mit Wandervorschlägen, Teil 2

Bei der Durchforstung der ersten 1024 Speicherzellen werden wir in dieser Folge die Adressen 3 bis 17 etwas genauer beleuchten.

Das letzte Mal hatten die besprochenen Speicherzellen verschiedene Bedeutungen für VC 20 und C 64.

Ab diesmal, also ab Speicherzelle 3 über mehrere Folgen dieser Serie hinaus bis zur Speicherzelle 672 gelten alle Angaben für beide Computer, 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 — $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 in der Tabelle 1 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, welche mit Gleitkommazahlen nicht so vertraut sind, möchte ich auf die Tabelle 2 verweisen. Er ist nur eine kleine Einführung. Später, bei der Behandlung der Speicherzellen 97 — 102 werde ich im Detail auf die externe und interne Darstellung und Verwendung von Gleitkommazahlen eingehen.

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.

Bis dahin haben Sie hoffentlich auch den Assemblerkurs weiter verfolgt, die Assembler-, Dissassembler- und Monitorprogramme eingetippt und können damit arbeiten. Dann können wir viel besser den ganzen Zusammenhang verfolgen.

Adresse 5 und 6 ($5 — $6)
Vektor auf die Routine zur Umwandlung einer ganzen Zahl in eine Gleitkommazahl

Dieses Programm ist die Umkehrung der oberen Routine. Es 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 benützt, 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 benützt. 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 vom Programmierer 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 (Semicolon !), 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 Semicolon. Dadurch erhöht sich laufend die in Speicherzelle 9 stehende Positionsangabe des Cursors. Wir erhalten folgenden Ausdruck:

```
. . X . 0 . . . X . 6 . . . X . 1 2 . . . X
. 1 9 . . . X . 2 6 . . . X . 3 3 . . . X .
4 0 . . . X . 4 7 . . . X . 5 4 . . . X . 6
1 . . . X . 6 8 . . . X . 7 5 . . . X . 8 2
. . . X . 1 . . . X . 7 . . . X . 1 3 . . .
```

Sie können die Positionsnummer nachrechnen. Berücksichtigen Sie bitte 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 ei-

nen 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, welche die Länge der Token-Zeile angibt.

Die Zelle 11 wird außerdem noch von den Basic-Routinen benützt, 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.

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 zuwenig 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 selbst definierte 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 nicht-identischen 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:

```
10 DATA 3
20 READ A
30 PRINT PEEK (17)
40 INPUT B
50 PRINT PEEK (17)
80 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)
Flagge für Vorzeichen des Ergebnisses bei SIN und TAN

Mit dieser Adresse fahren wir das nächste Mal fort.

(Dr. Helmuth Hauck/aa)

Tabelle 1. Was sind Zeiger, Vektoren und Flaggen?

Zeiger, Vektoren und Flaggen

Zeiger und Vektoren sind 2 benachbarte Speicherzellen (Bytes), die eine wichtige Adresse enthalten.
Wir sprechen von einem **Zeiger**, wenn diese Adresse den Beginn von gespeicherten Daten angibt.
Ein **Vektor** kennzeichnet den Beginn eines Maschinenprogramms. (Ich muß zugeben, daß diese Unterscheidung nicht immer scharf angewendet wird beziehungsweise anwendbar ist).
Eine **Flagge** besteht aus einer Zahl, die von einem Programm verwendet wird, um sich das Resultat einer Operation zu merken beziehungsweise für eine spätere Verwendung festzuhalten.

Tabelle 2. Die Zahlendarstellung bei den Commodore-Systemen

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: $^{2}/_{3}$, $^{26}/_{8}$, $^{15}/_{14}$
— 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 $^{51}/_{52}$ + $^{3}/_{4}$!!

Ohne lang zu überlegen, rechnen wir natürlich um, $^{51}/_{52}$ = 0,9807692 und $^{3}/_{4}$ = 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 (siehe auch Assembler Kurs im 64'er-Magazin, Ausgabe 11)