Assembler ist keine Alchimie

(Teil 6)

In der vorangegangenen Ausgabe haben wir die relative und die Zeropage-Adressierung kennengelernt. Heute kommt die indizierte Adressierung dran und natürlich sehen wir uns wieder einige neue Assembler-Befehle an. Wir werden uns einige Gedanken machen über die sogenannten Fließkommazahlen und den Basic-Befehl USR. Auch die Speicherorganisation unseres Computers soll uns nochmal beschäftigen.

Zunächst die indizierte Adressierung. Indizieren heißt, etwas mit einem Index, also einem Zeichen oder einer Nummer, zu versehen. Beispielsweise bezeichnet man in der Mathematik die beiden Lösungen einer quadratischen Gleichung häufig als X1 und X2. Dabei ist dann die Ziffer (1 oder 2) der Index und X ist eine indizierte Größe. Man geht also aus von einer festgelegten Grundmenge (Lösungsmenge X) und trifft durch den Index eine weitere Unterscheidung.

So ähnlich können wir uns auch die Funktion der indizierten Adressierung bei der Assembler-Programmierung vorstellen. Nehmen wir als Beispiel den Befehl
LDA 1500,X

Man spricht hier von einer absolut-X-indizierten Adressierung. Das Assemblerwort LDA ist uns bekannt: Lade den Akku. Woher soll der für den Akku bestimmte Inhalt geholt werden? Aus der Speicherzelle, die sich durch 1500 plus Inhalt des X-Registers ergibt. Steht also im X-Register zum Zeitpunkt des Befehlsaufrufes eine 5, dann wird der Akku aus Speicherzelle 1500+5, also 1505, geladen. Das X-Register kann Werte von 0 bis $FF (dez. 255) enthalten. Die Ähnlichkeit sieht also so aus:

Aus einer Gesamtmenge von 256 Adressen, die durch die Anfangsadresse (bei unserem Beispiel 1500) und die möglichen 256 Belegungen des X-Registers festgelegt sind (die Grundmenge), werden je nach X-Registerinhalt einzelne Adressen unterschieden und adressiert. Das X-Register fungiert dabei als ein Index, weswegen man auch oft die Bezeichnung »Index-Register X« in der Literatur findet.

Ebenfalls als Index-Register kann das Y-Register dienen, was zum Beispiel zum Befehl
LDX 1500,Y
führen kann. Dies ist dann eine absolut-Y-indizierte Adressierung.

Befehl	Indizierte Adressierung			
	absolut		Null-Seite-absolut	
	X	Y	X	Y
Folge 2				
LDA	+	+	+	—
LDX	—	+	—	+
LDY	+	—	+	—
STA	+	+	+	—
STX	—	—	—	+
STY	—	—	+	—
RTS	/	/	/	/
Folge 3				
INX	/	/	/	/
INY	/	/	/	/
INC	+	—	+	—
DEX	/	/	/	/
DEY	/	/	/	/
DEC	+	—	+	—
SED	/	/	/	/
CLD	/	/	/	/
BNE	/	/	/	/
Folge 4				
ADC	+	+	+	—
CLC	/	/	/	/
SBC	+	+	+	—
SEC	/	/	/	/
BEQ	/	/	/	/
BCC	/	/	/	/
BCS	/	/	/	/
BMI	/	/	/	/
BPL	/	/	/	/
BVC	/	/	/	/
BVS	/	/	/	/
Folge 5				
CMP	+	+	+	—
CPX	—	—	—	—
CPY	—	—	—	—
Folge 6				
BIT	—	—	—	—
CLV	/	/	/	/
NOP	/	/	/	/
TAX	/	/	/	/
TAY	/	/	/	/
TXA	/	/	/	/
TYA	/	/	/	/
JMP	—	—	—	—
JSR	—	—	—	—

\+ anwendbar
— nicht erlaubt
/ weder absolute noch Zeropage-Adressierung möglich

Tabelle 1. Anwendbarkeit der indizierten Adressierungsarten auf die bisher gelernten Assembler-Befehle.

Genauso wie man die normale absolute Adresse (also zum Beispiel 1500) als Basis der Indizierung durch das X- oder das Y-Register verwenden kann, ist das auch mit eine Zeropage-Adresse möglich. So gibt es zum Beispiel die Befehle
LDY 2B,X
oder
STX 19,Y

Man nennt diese Art der Adressierung dann Zeropage-absolut-X-indiziert beziehungsweise -Y-indiziert.

Weil die Zeropage aber nur 256 Adressen umfaßt, andererseits jedoch die Indexregister auch 256 Werte annehmen können, kann es geschehen (wenn man nicht aufpaßt), daß die Summe aus der Basisadresse (zum Beispiel $2B) und dem Indexregisterinhalt größer als 256 wird. Wenn zum Beispiel in dem Befehl
LDA FE,X
der X-Registerinhalt 2 beträgt, ergäbe sich $FE+$02=$0100. In diesem Fall wird aber nicht der Inhalt von $0100 in den Akku geladen, sondern der Befehl spricht die Speicherstelle $00 an. Der Grund dafür liegt in der Tatsache, daß unser Prozessor den Befehl als 2-Byte-Befehl interpretiert — das 2. Byte ist die Zeropageadresse, die sich als Summe ergibt — und deswegen nur das LSB der Adresse beachtet. Von $0100 ist das LSB aber $00. Mit anderen Worten: Die Zero-page-absolut-indizierten Befehle lassen einen Zugriff nur auf die Zeropage selbst zu. Dieses Verhalten muß man beim Programmieren beachten.

Wir wollen nochmal zusammenfassen. Vier neue Adressierungsarten haben wir kennengelernt:

Absolut-X-indiziert
zum Beispiel LDA 1500,X
Absolut-Y-indiziert
zum Beispiel LDA 1500,Y
Zero-page-absolut-X-indiziert
zum Beispiel LDA 2B,X
Zero-page-absolut-Y-indiziert
zum Beispiel LDX 2B,Y

Die Verwendung des Y-Registers als Indexregister ist stark eingeschränkt. Nur bei wenigen Befehlen ist sie erlaubt (tatsächlich nur LDX und STX bei Zeropage-absolut-indizierter Adressierung). In der Tabelle 1 sehen Sie, welche bisher behandelten Befehle wie mit der indizierten Adressierung verwendet werden dürfen.

Es gibt noch zwei weitere Arten einer indizierten Adressierung, auf die wir noch zu sprechen kommen werden.

Einige Nachzüglicher: Die Befehle BIT, CLV, NOP und TAX, TAY, TXA, TYA

Wir wollen noch ein bißchen aufräumen: Ein paar Befehle, die bisher zu keinem Gebiet so richtig paßten, sollen jetzt behandelt werden.

BIT: Dieser Befehl heißt »Bit-Test« und paßt von daher eigentlich zu den in der letzten Ausgabe behandelten Vergleichsbefehlen. Die Behandlung der Flaggen ist aber völlig anders. Nehmen wir das Beispiel
BIT 1500

Folgendes passiert: Der Inhalt der Speicherstelle $1500 wird mit dem Inhalt des Akku UND-verknüpft, das Ergebnis in der Z-Flagge angezeigt und Bit 7 sowie Bit 6 von $1500 in die N- beziehungsweise die V-Flagge übertragen. Weder Akku noch der Inhalt von $1500 verändern sich dabei.

Das ging ein bißchen holterdipolter. Sehen wir uns das jetzt mal ganz langsam Schritt für Schritt an! Zunächst die UND-Verknüpfung. Bit für Bit wird der Akku-Inhalt mit dem Inhalt der adressierten Speicherstelle UND-verknüpft. Dabei gelten folgende Regeln (die Leser der Grafik-Serie kennen das ja schon):
0 UND 0 = 0
0 UND 1 = 0
1 UND 0 = 0
1 UND 1 = 1

Nur dann also, wenn die entsprechenden Bits im Akku und in 1500 gleich 1 sind, ergibt sich bei der UND-Verknüpfung eine 1. Man stellt sowas meist in einer sogenannten Wahrheitstabelle zusammen (Tabelle 2).

UND	0	1
0	0	0
1	0	1

Tabelle 2. Wahrheitstabelle der logischen Verknüpfung UND

Nehmen wir als Beispiel mal an, im Akku stünde $0A und in der Speicherstelle $1500 wäre $09 enthalten. Die UND-Verknüpfung sieht dann so aus:
Akku $0A 0000 1010
1500 $09 0000 1001
UND ———————
0000 1000

Das Ergebnis ist also $08. In der Z-Flagge wird in dem Fall, daß das Ergebnis der UND-Verknüpfung ungleich Null ist (wie hier) eine Null angezeigt, sonst eine 1.

Wir haben in unserem Zahlenbeispiel mit dem BIT-Befehl überprüft, ob die Bits 1 und 3 in Speicherstelle $1500 gelöscht sind. Dazu haben wir in den Akku eine sogenannte Maske (hier also $0A) geladen. Das Ergebnis sagt uns, daß nicht beide Bits gelöscht waren. Wäre der Inhalt von $1500 beispielsweise $10 gewesen (0001 0010), hätten wir in der Z-Flagge eine 1 gefunden. Daher der Name »Bit-Test«: Durch geeignete Maskenwahl kann praktisch jedes Bit überprüft werden. Dabei werden weder der Akku-Inhalt noch der Inhalt der angesprochenen Speicherstelle verändert.

Der BIT-Befehl hat aber noch mehr Auswirkungen: Die Bits 6 und 7 der geprüften Speicherzelle findet man nach Befehlsausführung in zwei Flaggen nochmal:
Bit 7 in der N-Flagge
Bit 6 in der V-Flagge

Damit kann man beispielsweise überprüfen, ob sich am adressierten Ort eine negative Zahl befindet. Alle drei Flaggen können ja nun mit den Branch-Befehlen abgefragt werden. Sie erkennen sicherlich schon, wie vielseitig dieser merkwürdige BIT-Befehl einsetzbar ist.

Adressierbar ist BIT entweder absolut (wie im obigen Beispiel) oder Zeropage-absolut. Je nachdem liegt er dann als 3-Byte- oder als 2-Byte-Befehl vor.

CLV: Dieser Befehl heißt »clear overflow-flag«, also »lösche die Überlauf-Flagge«. Die V-Flagge war — wie Sie sich erinnern werden — unsere rote Ampel bei Rechenoperationen (siehe Ausgabe 12/84, S. 135). Es ist ein 1-Byte-Befehl mit impliziter Adressierung und interessant daran ist, daß es keinen Befehl gibt, der das Gegenteil — also das Setzen der V-Flagge — bewirkt.

NOP: NOP steht für »no operation«, was bedeutet »keine Tätigkeit«. Das ist der Nichtstu-Befehl. Er tut aber doch etwas: Er sorgt dafür, daß der Befehlszähler weitergezählt wird und bewirkt eine Verzögerung von 2 Taktzyklen. NOP ist ein 1-Byte-Befehl mit impliziter Adressierung. Er wird in fertigen Programmen nur selten verwendet: Zur Erzeugung einer kurzen definierten Verzögerung. Meist gebraucht man ihn bei der Erstellung eines Programmes als Platzhalter oder bei der Fehlersuche, um zum Beispiel unerwünschte Sprünge zu ersetzen.

Die Transporteure: TAX, TAY, TXA und TYA

Ab und zu ist es nötig, Registerinhalte untereinander auszutauschen. Viele Dinge (Addition, Subtraktion und so weiter) können nur im Akku geschehen. Wenn wir eine solche Operation beispielsweise mit dem Inhalt des X-Registers ausführen wollen, verschieben wir diesen Inhalt mit dem Befehl TXA. »Transfer X into accumulator« also »übertrage X-Register in den Akku« bedeutet das. Analog verwendet man TYA, um Y-Register-Inhalte in den Akku zu schieben oder für den umgekehrten Weg TAY beziehungsweise TAX (Akkuinhalt ins Y- beziehungsweise ins X-Register schieben). Genau genommen wird nicht übertragen, sondern nur kopiert: Die Register, aus denen verschoben wird, bleiben unverändert. Weil die jeweiligen Zielorte der Verschiebung (Akku, X- oder Y-Register) vom neuen Inhalt überschrieben werden, können sich auch Flaggen ändern. Betroffen sind von dieser Möglichkeit die N- und die Z-Flagge. Alle vier Befehle bestehen aus einem Byte und können natürlich nur implizit adressiert werden.

So springen die Assembler-Alchimisten: JMP, JSR

JMP und JSR entsprechen ungefähr den vom Basic her bekannten Befehlen GOTO und GOSUB.

JMP kommt von »jump to address«, also »springe zur angegebenen Adresse«. Nehmen wir uns wieder ein Beispiel vor:
JMP 1500
bewirkt einen Sprung zur Adresse 1500. Das funktioniert so: In den Programmzähler werden LSB und MSB der Zieladresse geladen. Das war dann auch schon der Sprung, denn der Programmzähler ist der Pfadfinder des Computers: Die Adresse, die dort steht, wird als nächste bearbeitet. Schalten Sie doch mal den SMON ein (oder einen anderen Monitor) und sehen Sie sich das mit folgenden Befehlen an:

```
1400 JMP 1500
```

Dort unterbrechen wird den Computer mit

```
1500 BRK
```

So weit, so gut: Wir starten mit dem SMON-Kommando G 1400 und erhalten eine Registeranzeige mit dem Programmzählerstand 1501. Genau das hatten wir ja erwartet.

Weniger durchschaubar ist das folgende Beispiel:

```
1400  LDA  #00
1402  LDX  #16
1404  STA  1300
1407  STX  1301
140A  JMP  (1300)
```

Dazu gehört dann noch die Programmzeile:

```
1600  BRK
```

Wenn Sie das genauso eingegeben haben und dann mittels G 1400 starten, erhalten Sie eine Registeranzeige mit dem Programmzählerstand 1601.

Schon an der neuen Schreibweise des Argumentes in Zeile 140A werden Sie bemerkt haben, daß hier nicht mehr die normale absolute Adressierung wie zuvor angewendet wird. Dies ist eine neue Form: Die **indirekte Adressierung**. Indirekt deswegen, weil wir nicht mehr direkt die Zieladresse angeben, sondern einen sogenannten Vektor. Ein Vektor besteht aus zwei aufeinander folgenden Speicherzellen (hier also 1300 und 1301), die in der Form LSB/MSB die eigentliche Zieladresse enthalten. Das LSB von $1600 ist $00. Das haben wir über den Akku nach $1300 geladen. Das MSB $16 kam durch das X-Register an seinen Platz $1301:

Zieladresse	16	00
	MSB	LSB
	↑	↑
Vektor	1301	1300

Das ist die Methode der toten Briefkästen, die in Kreisen der Assembler-Alchimisten anscheinend genauso beliebt ist wie bei Agenten. So wie diese im hohlen Baum die Treffpunktanschrift hinterlegt finden, verläßt sich unser Computer auf die Speicherstellen 1300 und 1301 für die Angabe der Zieladresse.

Diese Art der Adressierung ist im wahrsten Sinn des Wortes ein Unikum: Es gibt sie nämlich nur für den JMP-Befehl! Davon wird allerdings dann auch recht häufig Gebrauch gemacht, zum Beispiel im Betriebssystem unseres Computers. Aber darüber und über die Vektoren, die dazu verwendet werden, soll ein andermal berichtet werden.

Wir dürfen nämlich nicht den anderen **Sprungbefehl JSR** vergessen. JSR steht für »jump to subroutine«, was eingedeutscht etwa bedeutet »springe zum Unterprogramm«. Genauso wie in Basic Unterprogramme durch GOSUB ⟨Zeilennummer⟩ aufgerufen werden, kann das auch hier geschehen durch JSR Adresse. Hier ist nur die absolute Adressierung möglich. Das erste Beispiel soll uns zeigen, wie dieser Befehl funktioniert:

```
1400  JSR 1500
```

Dort soll dann erstmal stehen:

```
1500  BRK
```

Noch nicht starten!! Zunächst einmal verzeihen Sie mir diese Programmierer-Todsünde: Aus einem Unterprogramm heraus den Programmablauf zu beenden! Ich werd's auch nie wieder tun. Hier geschieht das nur zu

Lehrzwecken. Was läuft ab: Der Programmzählerinhalt plus 2 wird auf den Stapel gelegt und dann die Adressse 1500 in den Programmzähler geladen. Ebenso kurz wie unklar! Was ist denn ein Stapel? Also langsam, Schritt für Schritt.

Der Sinn von Unterprogrammen ist ja, daß der Computer nach Ende der Bearbeitung wieder ins aufrufende Hauptprogramm zurückkehrt. Er muß sich aber dazu irgendwo merken, von wo aus er zum Unterprogramm gesprungen ist. Dazu verwendet er den Stapel. Das ist ein Speicherbereich ($0100 bis $01FF), der direkt vom Prozessor aus verwaltet wird. Die genaue Architektur und Handhabung dieses »Prozessor-Stack« werden wir noch in einer späteren Folge kennenlernen. Uns soll hier nur interessieren, daß es einen Zeiger gibt, der auf den nächsten freien Platz im Stapel weist und daß dieser Speicher von oben nach unten gefüllt wird (wie in Basic bei den Strings). Wenn Sie mit Hilfe des SMON mal in den Stapel hineinsehen wollen, dann geben Sie doch mal ein M 0100 01FF. Was nun genau bei Ihnen drin steht, ist sehr von der vorherigen Nutzung Ihres Computers abhängig. Der Mikroprozessor nutzt den Stapel bei sehr vielen Tätigkeiten. Es kommt auch nur auf den Teil des Stapels an, der durch den Stapelzeiger als gefüllt bezeichnet wird. Der Stapelzeiger wird beim SMON in der Registeranzeige als SP angezeigt. Wenn Ihr Stapelzeiger (prüfen Sie das doch mal durch Eingabe von R) nun zum Beispiel F6 zeigt, dann bedeutet das, daß alle Stapelplätze von $01F6 an abwärts frei und die oberhalb bis $01FF besetzt sind. Beim Nachsehen mit M 01F0 01FF finden Sie dann beispielsweise:

```
:01F0  20 00 20 AA C1 FA C0 46
:01F8  E1 E9 A7 A7 79 A6 9C E3
```

Die Speicherstelle, auf die der Stapelzeiger weist, ist unterstrichen. Nun starten wir mit G 1400 unser kleines verbotenes Testprogramm. Es meldet sich die Registeranzeige. Im Stapelzeiger steht jetzt F4 (oder eben Ihr vorhergegangener SP minus 2). Wenn wir nun wieder im Stapel nachsehen mit M 01F0 01FF, dann finden wir im Gegensatz zur obigen Anuzeigè nun:

```
:01F0  20 AA C1 FA C0 02 14 46
                  "  "  "
:01F8  E1 E9 A7 A7 79 A6 9C E3
```

Unterstrichen ist wieder das Ziel des Stapelzeigers, der jetzt zwei Plätze weitergerückt ist, um der durch Pfeile gekennzeichneten Adresse 1402 (als LSB/MSB) Raum zu schaffen. $1402 ist das letzte Byte des JSR-Befehls. Wie wir den Programmzähler kennen, ist er im allgemeinen immer einen Schritt voraus. Hier liegt er aber einen zurück, falls er nach Beendigung des Unterprogrammes an der notierten Adresse weitermacht. Dazu kommen wir gleich noch. Was wir am Programmzähler aber auch noch nach Ablauf unseres kurzen Beispielprogrammes ablesen können, ist die Tatsache, daß die Sprungadresse 1500 in ihn geschrieben wird, somit der Sprung dann also stattgefunden hat.

Nun bauen wir das kleine Prögrämmchen etwas um:

```
1400    JSR 1500
1403    BRK
```

Das Unterprogramm soll nur aus dem Rücksprung bestehen:

```
1500    RTS
```

Verlangen Sie nun noch vor dem Start eine Registeranzeige mit R und merken Sie sich den Wert des Stapelzeigers. Dann starten Sie das Programm mit G 1400 und achten Sie auf die neue Registeranzeige. Zwei Dinge interessieren uns:

1) Der Wert des Stapelzeigers ist unverändert geblieben.
2) Der Programmzähler weist nun auf $1404.

Wenn Sie nun nochmal mit dem M-Befehl des SMON in den Stapel sehen, werden Sie unter Umständen zwar noch die Adresse 1402 dort finden (dann nämlich, wenn wir den Stapel seit dem letzten Programm nicht verändert haben). Wie Sie aber inzwischen wissen, hätte durch den neuen JSR-Befehl nochmal 1402 dort eingetragen sein müssen. Das stand da auch einige Mikrosekunden lang... bis der RTS-Befehl wirksam wurde. RTS macht ziemlich viel:

1) RTS holt die auf dem Stapel gespeicherte Adresse ab, und schreibt sie in den Programmzähler.
2) RTS vermindert dabei den Stapelzeiger um 2.
3) RTS addiert zum Programmzähler eine 1.

Deswegen kann das Programm also bei $1403 weiterlaufen und der Programmzähler nun hinter dem BRK-Befehl stehen.

Machen Sie doch mal etwas anscheinend total Verrücktes: Starten Sie mit G 1500. Es gibt da zwei Möglichkeiten, was geschehen kann: Entweder stand da noch vom ersten unterbrochenen Testprogramm die Adresse 1402. Dann endete nun alles mit einer Registeranzeige, bei der der Stapelzeiger um 2 höher gerutscht ist.

Oder da stand diese Adresse nicht mehr. Dann befinden Sie sich nun wieder im Basic. Wieso eigentlich? Als nächste Adresse finden Sie auf dem Stapel $E146 (dez.57670). Diese Adresse + 1 wird ja durch RTS in den Programmzähler gerufen. Ein Sprung an diese Adresse ist ein Sprung in ein Programm des Betriebssystems. Haben Sie ein ROM-Listing? Dann sehen Sie mal nach: Dort steht der Befehl...RTS. Dieses neuerliche RTS holt nun jedenfalls die nächste Adresse vom Stapel: $A7E9 (dez.42985). Diese Adresse + 1 im Programmzähler führt unseren Computer in die Basic-Interpreter-Schleife, also ins Basic zurück.

Wir haben so viel über den Stapel gehört, daß wir JSR fast schon wieder aus den Augen verloren haben. Deswegen nochmal eine kurze Übersicht:

a) JSR speichert den Programmzählerwert des letzten Bytes des Befehls auf dem Stapel
zum Beispiel 1402
b) stellt dabei den Stapelzähler um 2 zurück
zum Beispiel von $F6 nach $F4
c) schreibt in den Programmzähler die angegebene Zieladresse,
zum Beispiel 1500
d) Das Unterprogramm wird abgearbeitet bis der RTS-Befehl auftaucht.
e) Dann wird die gemerkte Adresse +1 in den Programmzähler geschrieben
zum Beispiel 1402+1=1403
f) und dabei der Stapelzähler wieder um 2 erhöht,
zum Beispiel von $F4 wieder zu $F6
g) Das Programm läuft nun wieder nach dem JSR-Befehl weiter,
zum Beispiel also bei 1403.

Nun sollte eigentlich auch klar sein, warum ein Aussprung aus einem Unterprogramm oder ein Abbruch im Unterprogramm eine Programmierer-Todsünde ist: Der Stapelzeiger wird nicht zurückgestellt. Die gemerkte Rücksprungadresse versauert allmählich auf dem Stapel. Noch schlimmer sind solche Sachen in einer Schleife, wo mehrfach aus dem Unterprogramm ausgebrochen wird: Hier ist der Stapel bald voll Müll und der Computer beendet seine Zusammenarbeit mit dem Programmierer. Weil aber Basic-Programme nichts anderes sind als eine Folge von Maschinenprogrammen, die je nach Befehl durch den Interpreter aneinandergereiht werden, ist das auch in Basic eine Todsünde. Wir wollen aber nicht so hart mit uns umgehen: Wenn wir gelernt haben, wie man mit speziellen Assembler-Befehlen im Stapel herumschaufeln kann, dann haben wir bei richtiger Anwendung von vorneherein jedenfalls in diesem Punkt die Absolution erhalten.

Alles fließt: Fließkommazahlen

Jeder, der tiefer in die Geheimnisse der Assembler-Alchimie eindringen will, muß sich vertraut machen mit der häufigsten Art der Zahlenverarbeitung in unserem Computer. Das ist die Handhabung von Fließkommazahlen (auch Gleitkommazahlen genannt). Wir werden dazu folgende Fragen zu klären haben:

1) Was sind Fließkommazahlen?
2) Wie sehen sie im binären Zahlensystem aus?
3) Wie behandelt unser Computer positive und negative Fließkommazahlen?
4) Wie können wir als Programmierer Einfluß nehmen auf die Verarbeitung dieser Zahlen im Computer?

Die Behandlung dieser vier Fragen wird uns eine ganze Weile beschäftigen. Fangen wir mit der ersten an: In Standardwerken der Mathematik werden Sie lange suchen müssen, um den Begriff »Fließkommazahl« zu finden. Im deutschen Sprachraum gibt es häufiger die Bezeichnung »wissenschaftliche Zahlendarstellung«. Das klingt sehr hochgestochen und ist eigentlich ganz einfach. Leser der Grafik-Serie werden sich vielleicht noch erinnern: Die Zahl 1000 kann man auf verschiedene Weise darstellen:

1000 = 10 * 10 * 10 = 10^3 ect.

Die hochgestellte Zahl (in Computerschreibweise: Die Zahl hinter dem Hochpfeil) ist hier gleich der Anzahl der Stellen minus 1 (1000 hat vier Stellen, also ist die Hochzahl eine 3). Diese Hochzahl nennt man Exponent (vom lateinischen exponere = anzeigen, herausheben). Nehmen wir nun einige andere Zahlen:

200 = 2 * 100 = 2 * 10^2
oder
2500 = 2,5 * 1000 = 2,5 * 10^3

Ich glaube, jetzt beginnt es Ihnen klarzuwerden, daß man auf diese Art wohl alle Zahlen irgendwie darstellen kann. Man dröselt die Zahlen auseinander, bildet ein Produkt, von dem der eine Multiplikator durch 10 teilbar ist (durch die Basis unseres normalen Zahlensystems). Genauer gesagt: Ein Faktor (also in den Beispielen 1000 oder 100) ist darstellbar als Potenz von 10. Der andere Faktor (in den Beispielen 1 oder 2 oder 2,5) wird Mantisse (vom lateinischen manitissa = Zugabe, Anhang, Schleppe) genannt. Sehen wir uns nochmal 2500 an:

```
2500 =  2,5 * 1000  =  2,5 * 10^3
     =   25 *  100  =   25 * 10^2
     =  250 *   10  =  250 * 10^1
     = 2500 *    1  = 2500 * 10^0
```

Das letzte war nur der Vollständigkeit halber, denn irgendeine Zahl hoch 0 ist immer 1. Man kann auch aus der 2500 folgendes machen:

2500 = 0,25 * 10000 = 0,25 * 1014
oder = 0,025 * 100000 = 0,025 * 1015
und so weiter. Oder anders herum:
2500 = 25000 * 0,1 = 25000 * 10↑—1
= 250000 * 0,01 = 250000 * 10↑—2
und so weiter.

Dabei bedeutet:
10↑—2 = 1/10↑2 = 0,01

Man kann sich das merken, indem man die Anzahl der Stellen zählt, um die man das Komma verschiebt. Diese Anzahl addiert man dann zur Hochzahl. Zur Erläuterung:
0,12345 = 1,2345 * 10↑—1

Wir haben das Komma um eine Stelle nach rechts gerückt, weshalb wir die Hochzahl —1 schreiben müssen (vorher war da nämlich unsichtbar die Hochzahl 0: und 10↑0 = 1).
0,12345 = 123,45 * 10↑—3

Hier wurde das Komma um drei Stellen nach rechts verschoben. Daher der Exponent —3. Sie sehen folgenden Zusammenhang:

Komma eine Stelle nach rechts verschoben: Exponent + (—1).
Zum Beispiel
0,1234*10↑—2 = 1,234*10↑—3

Komma eine Stelle nach links verschoben: Exponent + 1.
Zum Beispiel
3,14*10↑2 = 0,314*10↑3

Verstehen Sie nun, warum man diese Art der Zahlendarstellung Fließkomma- oder Gleitkommazahlen nennt?

Vielleicht sehen Sie aber noch nicht den Sinn der Fließkommazahlen ein. Dazu gebe ich Ihnen zwei einsichtige Beispiele. Der Atomkern eines Heliumatoms wiegt etwa (halten Sie sich fest): 0,000 000 000 000 000 000 000 000 006 643 kg.

Sehr unbequem, diese ganzen Nullen immer mitzuschleppen. Wir verschieben deshalb das Komma um 27 Stellen nach rechts und schreiben dann
6,643*10↑—27 kg.

2. Beispiel: Wir haben einen Ballon mit diesem Gas gefüllt. Bei normalen Temperatur- und Luftdruckbedingungen befinden sich in einem Kubikzentimeter im Ballon ungefähr (nochmal festhalten!):
26 900 000 000 000 000 000 Heliumatome

Wieder eine recht unangenehme Nullschlepperei. Wir verschieben das Komma um 19 Stellen nach links und erhalten 2,69*10↑19 Heliumatome. Fein, nicht wahr!

Abgesehen von der höheren Bequemlichkeit: Der Computer müßte allerhand Speicherplatz zur Handhabung der vielen Nullen bereitstellen. Mit BCD-Zahlen könnten wir zwar jede Zahl erfassen, hätten aber immer unterschiedlich viele Bytes zu verarbeiten. Wenn wir Fließkommazahlen verwenden, können wir — wie Sie noch sehen werden — jede (na sagen wir mal: fast jede) Zahl in der gleichen Anzahl Bytes aufbewahren.

Vom Basic her kennen Sie Fließkommazahlen auch (hier wird das Komma allerdings durch den Punkt ersetzt, entsprechend der angloamerikanischen Schreibweise). Das sind die, wo man zum Beispiel schreibt 6.02E23 oder 6.02E+23, was dann bedeutet 6,02*10↑23. E steht dort für Zehnerexponent. Durch die Art, wie Fließkommazahlen im normalen Computerdasein gespeichert werden, ergeben sich obere und untere Grenzen. Die höchste in Basic verarbeitbare Zahl im C 64 ist
+1.70141183*10E38

Größere Zahlen verursachen in Basic einen OVERFLOW ERROR. Was in Maschinensprache mit größeren Zahlen geschieht, ist weitgehend unsere Sache. Die dem Betrag nach kleinste verarbeitbare Zahl ist
± 2.93873588*10E—39

In Basic arbeitet bei Unterschreitung der Computer einfach mit einer Null weiter. Für die Behandlung in Maschinensprache sind ebenfalls wir als Programmierer verantwortlich.

Für diesmal sei's genug der Zahlenspiele: In der nächsten Ausgabe werden wir uns weiter mit Fließkommazahlen befassen.

Die USR-Funktion

Wieder einmal soll uns das Zusammenspiel von Basic und Maschinensprache beschäftigen. Einen Aufruf von Maschinenroutinen — nämlich den mit SYS — haben wir schon kennengelernt. Wir POKEten die zu übergebenden Werte an die Abrufspeicherstellen. Bei diesen Werten hat es sich um einfache Integerzahlen gehandelt, zum Beispiel die Anzahl der Glieder einer zu summierenden arithmetischen Reihe. Was tun wir aber, wenn wir Fließkommavariable an ein Maschinenprogramm übermitteln wollen? Gewiß, werden Sie sagen, lernen wir das ja in den nächsten Folgen und können dann entsprechende POKE-Kommandos geben. Damit haben Sie auch recht, nur ist das dann der »harte« Weg. Es gibt auch einen problemlosen »weichen« Weg, nämlich das USR-Kommando.

USR ist ein Basic-Befehl und rührt her von »User callable machine language subroutine«, also »durch den Benutzer aufrufbares Maschinensprachunterprogramm«. Darin liegt eigentlich noch nichts Neues gegenüber dem SYS-Befehl. Im Gegensatz zu SYS — wo das Argument die Einsprungadresse des Maschinenprogrammes ist — übergibt USR als Argument eine beliebige Fließkommavariable in festgelegter Form an eine sehr nützliche Speicherstellenkombination, den Fließkomma-Akkumulator 1, von uns künftig einfach FAC genannt. Der FAC belegt die Speicherstellen 97 bis 102 ($61 bis $66). Wenn das eventuell in Basic benötigte Ergebnis dort auch in der vorgeschriebenen Form abgelegt wird, kann es im Basic-Programm weiterverwendet werden. Keine Angst, dazu kommen wir bei der weiteren Behandlung der Fließkommazahlen noch ganz ausführlich zu sprechen. Heute soll uns das noch nicht belasten. Als Argument kann man nämlich auch irgendeine bedeutungslose Größe, ein sogenanntes Dummy angeben, das dann gar nicht weiter verwendet wird. Der USR-Befehl dient in diesem Fall lediglich dem bequemen Ansteuern eines Maschinenprogrammes.

Woher weiß unser Computer beim USR-Befehl, welche Maschinenroutine er im 64-KByte-Speicher bearbeiten soll? Beim SYS-Befehl ist das klar: Das Argument sagt es:
SYS 24345
läßt den Programmzähler auf dez.24345 zeigen. Aber wenn wir eingeben:
USR(24345)
dann packt der Computer die Zahl 24345 als Fließkommavariable in den FAC und meldet dann einen SYNTAX ERROR. Das liegt daran, daß der Basic-Interpreter beim USR-Befehl einen der oben kennengelernten indirekten Sprünge vollführt:
JMP (311)

$311/312 (in dezimal 785/786) ist also ein Vektor, und der weist im Normalfall zu einer Routine, die den SYNTAX ERROR ausgibt (dez. 45640). Bevor wir also den USR-Befehl geben, müssen wir in diesen Vektor die Startadresse unserer Maschinenroutine schreiben:

dez. 24345 = $5F19
LSB $19 = dez. 25 in Speicher 785 mit POKE 785,25
MSB $5F = dez. 95 in Speicher 786 mit POKE 786,95

Jetzt weiß der Computer, wohin er beim USR-Aufruf springen soll, und solange, bis wir den Vektor wieder ändern, führt er bei jedem USR-Befehl unser bei 24345 stehendes Maschinenprogramm aus. Wir müssen nur noch dafür sorgen, daß dort dann auch wirklich eines anfängt. Ein Beispiel werden wir nachher noch behandeln.

Der harte Kern: Nochmal Speicherfragen

Die Struktur des C 64-Speichers ist vereinfacht schon in der Grafik-Serie und zu Beginn dieses Kurses gezeigt worden. Dabei tauchten zwei ROM-Bereiche auf, die wir Basic-Interpreter und Betriebssystem genannt haben. Diese Unterteilung ist nicht ganz korrekt. Wenn Sie über ein ROM-Listing verfügen und beispielsweise das Ende des ROM-Bereiches von $A000 bis $BFFF sowie den Anfang des oberen ROM ($E000 bis $FFFF) untersuchen, dann stellen Sie fest, daß ab dez. 49087 ($BFBF) die Basic-Funktion EXP bearbeitet wird. Der letzte Befehl vor $C000 beendet diese Funktion aber nicht etwa, sondern dort steht:
JMP E000

Tatsächlich läuft ab $E000 bis $E042 die Bearbeitung der EXP-Funktion munter weiter, und auch danach finden sich allerlei Basic-Befehle (SIN, COS und so weiter). Da liegt also keine klare Trennung vor, sondern ein Mischmasch. Wir sollten uns vielleicht angewöhnen — statt vom Interpreter und dem Betriebssystem —, vom unteren und oberen ROM-Bereich zu sprechen.

Eine andere Unterscheidung ist dagegen sinnvoll: Wie einige Besitzer neuerer Commodore 64 sicherlich bemerkt haben, sind Teile der ROM-Routinen im Laufe der Zeit verändert worden. Hauptsächlich geht es bei den aktuellen Neuerungen dieser internen Maschinenprogramme um die Farbgebung der Zeichen. Man kann eigentlich nie so recht wissen, was den Software-Planern von Commodore noch alles einfällt. Jedenfalls können deren Ideen manchmal recht dramatische Folgen haben, nämlich dann, wenn Sie ein fabelhaftes Maschinenprogramm gebaut haben, welches ROM-Routinen direkt verwendet. Der Programmierer spielt auf diese Weise eine milde Form des russischen Roulette. Glücklicherweise halten sich die Änderungen in Grenzen, und wir dokumentieren unsere Programme ja auch immer gut (Sie etwa nicht??). Notwendige Umbauten können also leicht vonstatten gehen.

Ganz ohne ROM-Routinen-Verwendung kommt man eigentlich kaum aus. Es gibt aber einen ROM-Bereich, für den Commodore verspricht, keinerlei Änderungen durchzuführen: die KERNAL-Sprungtabelle.

Das ist ein Programmbereich ($FF81 bis $FFF5), in dem 39 JMP-Befehle enthalten sind (zum Teil in absoluter, aber auch in indirekter Adressierung). Jeder dieser Sprungbefehle weist auf die Einsprungadresse eines Maschinenprogrammes. Da finden sich alle wichtigen Ein/Ausgabe-Operationen, Systemtakt- und Uhrsteuerungen und anderes mehr. Wir werden uns nach und nach damit vertraut ma-

chen. In der Tabelle 3 sind die KERNAL-Adressen und ihre Funktion aufgeführt. Manche davon können ohne jede Vorbereitung benutzt werden, andere brauchen bestimmte Routinen oder Angaben, um sinnvoll zu arbeiten.

Die Absicht von Commodore ist es, daß jeder Aufruf von zum Beispiel FFD2 die Ausgabe eines Zeichens bewirkt, und zwar unabhängig davon, welchen Computer in welcher Version wir benutzen. Das Programm, welches diese Zeichenausgabe letztendlich ausführt, kann sich ändern, kann in ganz andere Speicherbereiche gelegt werden. An der Stelle $FFD2 wird aber immer ein JMP mit der Einsprungadresse stehen. Leider ist diese Sprungtabelle viel zu knapp gehalten. Es gibt so viele interessante ROM-Routinen, die wir alle ohne diese schöne Sicherheit anspringen müssen.

Adresse HEX	Adresse dezimal	Name	Funktion
FF81	65409	CINT	Prüfen der TV-Norm, Berechnung der Taktfrequenz
FF84	65412	IOINIT	Ein/Ausgabe-Reset
FF87	65415	RAMTAS	Prüfen auf freien Basic-RAM
FF8A	65418	RESTOR	Initialisieren der I/O-Vektoren
FF8D	65412	VECTOR	Lesen und Setzen der I/O-Vektoren
FF90	65424	SETMSG	Setzen des Ausgabe-Modus
FF93	65427	SECOND	Ausgeben der Sekundäradresse nach LISTEN
FF96	65430	TKSA	Ausgabe der Sekundäradresse nach TALK
FF99	65433	MEMTOP	Lesen/Setzen des Speicherendes
FF9C	65436	MEMBOT	Lesen/Setzen des Speicheranfangs
FF9F	65439	SCNKEY	Abfragen der Tastatur
FFA2	65442	SETTMO	Setzen der Time-Out-Flagge
FFA5	65445	ACPTR	Zeichen vom seriellen Port in Akku lesen
FFA8	65448	CIOUT	Zeichen vom Akku auf seriellen Port ausgeben
FFAB	65451	UNTLK	Sendet UNTALK an seriellen Bus
FFAE	65454	UNLSN	Sendet UNLISTEN an seriellen Bus
FFB1	65457	LISTEN	Sendet LISTEN an Geräte per seriellen Bus
FFB4	65460	TALK	Sendet TALK an Geräte per seriellen Bus
FFB7	65463	READST	Liest I/O-Status in den Akku
FFBA	65466	SETLFS	Festlegung der Parameter für OPEN
FFBD	65469	SETNAM	Festlegung des Filenamens
FFC0	65472	OPEN	Öffnet spezifizierten File
FFC3	65475	CLOSE	Schließt spezifizierten File
FFC6	65478	CHKIN	Öffnet einen Eingabekanal
FFC9	65481	CHKOUT	Öffnet einen Ausgabekanal
FFCC	65484	CLRCHN	Schließt Ein- und Ausgabekanäle
FFCF	65487	CHRIN	Holt vom aktiven Eingabekanal ein Zeichen in den Akku
FFD2	65490	CHROUT	Sendet Akku-Inhalt auf aktiven Ausgabekanal
FFD5	65493	LOAD	LOAD und VERIFY von Programmen
FFD8	65496	SAVE	Speichern von Programmen
FFDB	65499	SETTIM	Uhrzeit setzen
FFDE	65502	RDTIM	Uhrzeit lesen
FFE1	65505	STOP	STOP-Taste abfragen
FFE4	65508	GETIN	Zeichen aus dem Tastaturpuffer in den Akku lesen
FFE7	65511	CLALL	Schließen aller Kanäle und Files
FFEA	65514	UDTIM	Uhr um 1/60 Sekunde weiterzählen
FFED	65517	SCREEN	Lesen des Bildschirmformates
FFF0	65520	PLOT	Lesen/Setzen der Cursor-Position
FFF3	65523	IOBASE	Lesen der Startadresse der Ein- und Ausgabebausteine

Tabelle 3. Kernal-Routinen

Die Urzelle eines Programmprojektes

Wir sind jetzt soweit, daß wir die Urzelle eines Programmprojektes, welches uns eine lange Zeit begleiten wird, aufbauen können. Wir wollen etwas unter den Teppich kehren. Der Teppich, das sind die uns bislang nicht zugängigen RAM-Bereiche unter den ROMs. Haben Sie das nicht auch schon mal erlebt, daß Sie während einer Programmarbeit plötzlich feststellen, Sie benötigen zum Beispiel für eine Zwischenrechnung ein weiteres Programm, oder Sie wälzen Listen und denken sich, ein kleiner Hilfsbildschirm wäre jetzt von Nutzen, oder....

Mit diesem heute zu startenden Programm wäre all das und noch viel mehr realisierbar. Es soll auf einfache Weise beliebige Speicherbereiche unters ROM schieben und sie wieder hervorholen können.

Natürlich braucht die Entwicklung dieses Projektes einige Zeit, zumal wir noch vieles lernen müssen. Deswegen sind wir in dieser ersten Urzelle noch sehr eingeschränkt: Wir verschieben zuerst einmal nur eine Bildschirm-Kopfzeile unter den oberen ROM-Bereich. Auch in dieser einfachsten Version gibt es noch einige Programmteile, die Sie erst nach der nächsten Ausgabe verstehen werden. Aber irgendwann müssen wir ja mal anfangen, Nägel mit Köpfen zu machen.

Unser Maschinenprogramm soll durch die USR-Funktion aufgerufen werden. Wie wir es in dieser Ausgabe gelernt haben, muß deshalb vor dem ersten Aufruf eine Initialisierung durch Belegen des USR-Vektors mit unserer Startadresse stattfinden. Die Startadresse soll $02B6 (dez. 694) sein, denn dort gibt es einen freien RAM-Bereich bis inklusive $02FF (dez. 767), der weder andere Programme noch Kassettenoperationen stört. Das MSB $02 ist dezimal auch 2 und wird nach 786 gePOKEt:

POKE786,2

Das LSB $B6 ist dezimal 182 und soll in 785 geschrieben werden:

POKE785,182

Damit ist der USR-Vektor gestellt und wir brauchen uns nicht mehr weiter darum zu kümmern: Jeder USR-Aufruf wird nun den Start des Programmes bewirken. Nun zum Programm selbst. In Bild 1 finden Sie ein Flußdiagramm dazu.

Zunächst konstruieren wir den Teil, der die erste Bildschirmzeile nach $E000 und folgende Speicherstellen schiebt. Das X-Register verwenden wir als Index und laden es mit dez.40 = $27.

Schalten Sie also den SMON ein und starten Sie den Assembler mit:

A 02B6

Dann geben Sie ein:

```
02B6    LDX #27
```

Nun packen wir das letzte Zeichen der obersten Bildschirmzeile in den Akku:

```
02B8    LDA 0400,X
```

In das Y-Register legen wir die dazugehörige Farbe aus dem Bildschirmfarbspeicher:

```
02BB    LDY D800,X
```

Den Akkuinhalt — also die Bildschirminformation — legen wir nach $E000+$27:

```
02BE    STA E000,X
```

Dasselbe tun wir mit dem Farbcode, der ab $E028+$27 abwärts gespeichert wird. Leider kann man STY nicht X-indiziert-absolut adressieren (siehe Tabelle 1). Deshalb schieben wir zuerst den Y-Registerinhalt in den Akku:

```
02C1    TYA
02C2    STA E028,X
```

Damit ist das letzte Zeichen der Kopfzeile verschoben. Wir zählen das X-Register um 1 herunter:

```
02C5    DEX
```

Der X-Index weist nun auf das vorletzte Zeichen, mit dem sich alles ab $02A9 wiederholt. Wenn das X-Register bis 0 heruntergezählt ist, weist es auf das erste Zeichen der Kopfzeile. Die Schleife muß dann noch einmal durchlaufen werden und ein weiteres Herabzählen des X-Registers erzeugt $FF, was zum Setzen der N-Flagge führt. Das ist dann unser Signal, daß die gesamte Kopfzeile übertragen wurde. Die N-Flagge wird durch den BPL-Befehl getestet:

```
02C6    BPL 02B8
```

So weit, so gut. Wir hätten natürlich auch das X-Register von 0 an hochzählen können. Zum Beenden der Schleife wäre dann aber ein CPX-Befehl erforderlich gewesen, der jedesmal den X-Registerinhalt mit der Zahl $27 vergleicht.

MERKE: Indexregister in Schleifen abwärts zu zählen, kann Rechenzeit einsparen!

Ab $02CE soll der umgekehrte Vorgang, also das Zurückschieben der vorher gespeicherten Kopfzeile in den Bildschirmspeicher geschehen. Das einfachste wäre es sicherlich, diesen Programmteil mit einem weiteren USR-Kommando zu starten. Das sähe dann so aus:

1.USR-Befehl – schiebt Kopfzeile unter oberes ROM
2.USR-Befehl – holt Kopfzeile zurück in Bildschirmspeicher
3.USR-Befehl – schiebt wieder Kopfzeile unter ROM
4.USR-Befehl – holt sie wieder zurück und so weiter.

Weil aber das Umstellen des USR-Vektors durch POKEs vom Basic aus lästig ist, tun wir das einfach immer am Ende des betreffenden Maschinenprogrammabschnittes. Wir schreiben also das LSB der Programmfortführung ($CE) nach $311. Das MSB bleibt unverändert $02.

```
02C8    LDA #CE
02CA    STA 0311
02CD    RTS
```

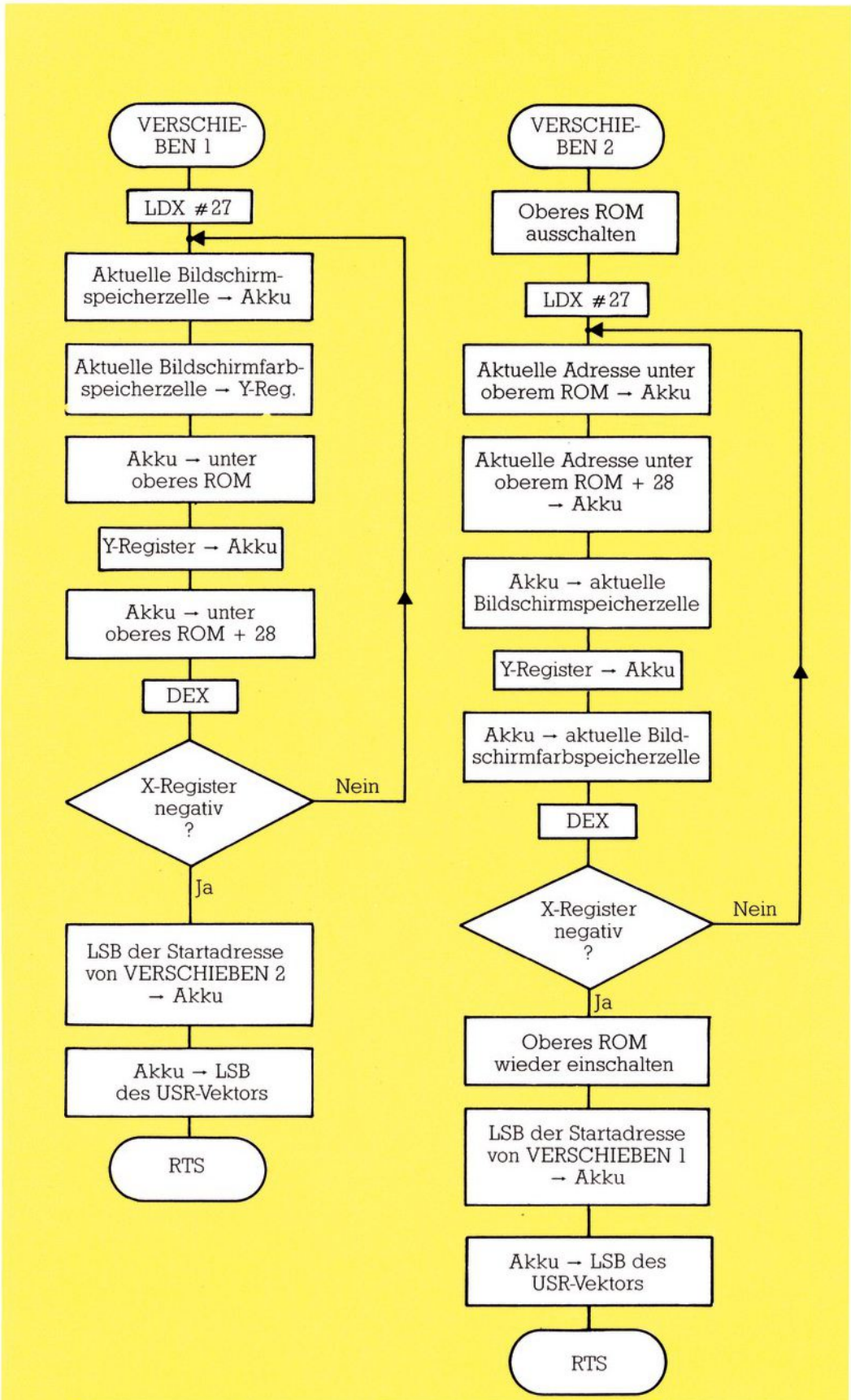

Bild 1. Das Flußdiagramm zu dem im Text erklärten Programm.

Mit dem RTS sind wir wieder im Basic-Programm gelandet, welches nun normal weiterverarbeitet wird. Erst ein neues USR-Kommando — im Programm oder im Direktmodus — startet den zweiten Teil unseres Maschinenprogrammes (weil in $0311, — der Einsprungpunkt des USR-Befehls — die Startadresse der auszuführenden Routine steht).

Einfache Befehle mit großer Wirkung

In diesem 2. Teil müssen wir erst einige Befehle geben, die Sie jetzt vielleicht noch nicht verstehen. Das hängt damit zusammen, daß zum Herauslesen des RAM unter dem ROM das ROM ausgeschaltet werden muß (entspricht POKE 1,53):

```
02CE    LDA 01
02D0    PHA
02D1    LDA #35
02D3    STA 01
```

(Der PHA-Befehl dient hier zur Zwischenspeicherung des Akku-Inhaltes). Das ist hiermit geschehen und wir kommen wieder in bekannte Gefilde mit der Ausleseschleife:

```
02D5    LDX #27
02D7    LDY E000,X
02DA    LDY E028,X
02DD    STA 0400,X
02E0    TYA
02E1    STA D800,X
02E4    DEX
02E5    BPL 02D7
```

Damit ist die gesamte gespeicherte Kopfzeile wieder zurückgeholt und wir können das ROM wieder einschalten:

```
02E7    PLA
02E8    STA 01
```

Falls nun wieder ein USR-Kommando auftaucht, soll die Kopfzeile mit dem 1. Programmteil unter das obere ROM gelegt werden wie am Anfang. Wir müssen deshalb den USR-Vektor auf $02B6 zurückschreiben:

```
02EA    LDA #B6
02EC    STA 0311
02EF    RTS
```

Das wärs! Wenn nun im Programm oder im Direktmodus wieder ein USR-Befehl auftritt, kann das Ganze von vorne beginnen. In dieser Version wird jedesmal eine neue Kopfzeile hin- und wieder zurückgeschoben. Wenn Sie eine einmal festgelegte Kopfzeile immer wieder benutzen möchten, dann stellen Sie den USR-Vektor einfach nicht mehr zurück: Lassen Sie also die Befehle bei 02EA und 02EC weg. Das Programm endet in dem Fall mit:

```
02EA    RTS
```

```
1 REM ********************************    <250>
2 REM *                              *    <229>
3 REM *  TEST FUER DIE 1. VERSION DES *    <139>
4 REM *       PROGRAMM-PROJEKTES     *    <048>
5 REM * V E R S C H I E B E N  V O N *    <009>
6 REM *       SPEICHERBEREICHEN      *    <193>
7 REM *                              *    <234>
8 REM * HEIMO PONNATH    HAMBURG 1984 *    <081>
9 REM ********************************    <002>
10 REM                                     <153>
15 REM ++++++ USR-VEKTOR EINSTELLEN ++++   <065>
20 REM                                     <163>
25 POKE 785,182:POKE 786,2                 <239>
30 REM                                     <173>
35 REM ++++++ KOPFZEILE +++++++++++++++++  <013>
40 REM                                     <183>
45 PRINT CHR$(147)CHR$(18)"TEST
   : BILD $0400=1024, FARBE $D800=55296"CHR$(14
   6)                                      <071>
50 PRINT:PRINT:PRINT"DURCH IRGENDEIN USR-KOMMAN
   DO WIRD NUN IM PROGRAMM-MODUS"          <020>
55 PRINT"DER ERSTE TEIL DES VERSCHIEBE-PROGRAMM
   ES AUFGERUFEN"                          <110>
60 PRINT"DIE KOPFZEILE WIRD UNTER DAS OBERE
    ROM{2SPACE}KOPIERT."                   <215>
65 REM                                     <208>
70 REM ++++++ 1. USR-AUFRUF ++++++++++++   <042>
75 REM                                     <218>
80 A=USR(1)                                <124>
85 PRINT:PRINT"HIER GESCHIEHT DAS DURCH A=USR(1
   ) IN{4SPACE}ZEILE 65"                   <132>
90 PRINT"DABEI IST 1 EIN DUMMY UND MIT A FANGEN
   {2SPACE}WIR AUCH NICHTS WEITER AN."     <063>
95 PRINT"AUF TASTENDRUCK WIRD DER BILDSCHIRM
   {2SPACE}GE-LOESCHT"                     <029>
100 REM                                    <243>
105 REM ++UEBERSCHREIBEN DER KOPFZEILE ++  <046>
110 REM                                    <253>
115 POKE 198,0:WAIT 198,1:PRINT CHR$(147)  <090>
120 REM                                    <007>
125 REM +++ NEUBEGINN DES PROGRAMMES ++++  <173>
130 REM                                    <017>
135 PRINT CHR$(19)"WAS AUCH IMMER JETZT IN DER
    KOPFZEILE{3SPACE}STEHT, ES WIRD BEIM 2.USR"
                                           <017>
140 PRINT"VON DEM ZUVOR DURCH DAS ERSTE USR
    GE-{3SPACE}SPEICHERTE UEBERSCHRIEBEN"  <078>
145 PRINT:PRINT"WENN SIE JETZT EINE TASTE DRUEC
   KEN..."                                 <104>
150 POKE 198,0:WAIT 198,1                  <246>
155 REM                                    <042>
160 REM ++++++ 2. USR-AUFRUF ++++++++++++  <090>
165 REM                                    <052>
170 A=USR(1):PRINT                         <169>
175 PRINT"IST DIE ALTE KOPFZEILE ZURUECK IN
    DEN{3SPACE}BILDSCHIRMSPEICHER GESCHOBEN."
                                           <164>
180 END                                    <052>
```

Listing 1. Test und Demonstration der Verschieberoutine. Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
LDA	absolut,X	3	BD	189	4	N,Z
	0-page-abs,X	2	B5	181	4	N,Z
LDX	absolut,Y	3	B9	185	4	N,Z
	absolut,Y	3	BE	190	4	N,Z
LDY	0-page-abs,Y	2	B6	182	4	N,Z
	absolut,X	3	BC	188	4	N,Z
STA	0-page-abs,X	2	B4	180	4	N,Z
	absolut,X	3	9D	157	5	N,Z
	absolut,Y	3	99	153	5	/
STX	0-page-abs,X	2	95	149	4	/
STY	0-page-abs,Y	2	96	150	4	/
INC	0-page-abs,X	2	94	148	4	/
	absolut,X	3	FE	254	7	/
DEC	0-page-abs,X	2	F6	246	6	N,Z
	absolut,X	3	DE	222	7	N,Z
ADC	0-page-abs,X	2	D6	214	6	N,Z
	absolut,X	3	7D	125	4	N,Z
	absolut,Y	3	79	121	4	N,V,Z,C
SBC	0-page-abs,X	2	75	117	4	N,V,Z,C
	absolut,X	3	FD	253	4	N,V,Z,C
	absolut,Y	3	F9	249	4	N,V,Z,C
CMP	0-page-abs,X	2	F5	245	4	N,V,Z,C
	absolut,X	3	DD	221	4	N,V,Z,C
	absolut,Y	3	D9	217	4	N,Z,C
BIT	0-page-abs,X	2	D5	213	4	N,Z,C
	absolut	3	2C	44	4	N,Z,C
CLV	0-page-abs.	2	24	36	4	N,V,Z
NOP	implizit	1	B8	184	3	N,V,Z
TAX	implizit	1	EA	234	2	V
TAY	implizit	1	AA	168	2	/
TXA	implizit	1	A8	138	2	N,Z
TYA	implizit	1	8A	152	2	N,Z
JMP	absolut	3	98	76	2	N,Z
	indirekt	3	4C	108	3	N,Z
JSR	absolut	3	6C	32	5	/
			20		6	/

Tabelle 4. Zusammenfassung aller wichtigen Daten der neuen Befehle

Eine wichtige Bemerkung noch: So bequem der Ort auch ist, an dem unser kurzes Programm steht, er hat einen gravierenden Nachteil: Falls Sie mittels einer RESET-Taste oder per Software einen Basic-Kaltstart durchführen, geht unser Programm flöten! Dieser Speicherbereich wird im Reset-Programm nämlich mit lauter Nullen überschrieben. Deswegen speichern Sie es bitte bald ab.

Damit sind wir für diesmal am Ende. Sie finden noch als Listing 1 ein kleines Testprogramm für unsere Verschieberoutine, und in Tabelle 4 wie immer, eine Zusammenfassung aller wichtigen Daten der neuen Befehle. In der nächsten Folge greifen wir noch einmal das Thema Fließkomma auf, werden die einfachsten und kürzesten Kurzspeicher-Befehle kennenlernen und beginnen mit den leistungsfähigsten Befehlen des 6502, den indirekt-indizierten-Befehlen.

(H. Ponnath/gk)