Tips & Tricks ausführlich erklärt

Die folgenden Programmbeispiele sollen vor allem dem Anfänger den Einstieg in die Maschinensprache des C 64 erleichtern.

Zu ihrem Verständnis sollte man wenigstens ungefähr mit dem Befehlssatz der 6510-CPU und mit der Speicherorganisation des C 64 vertraut sein. Die Beispiele stammen aus den verschiedensten Anwendungsgebieten. Ihnen allen gemeinsam ist:
- eine überschaubare Kürze
- Formulierung als Hypra-Ass-Quelltext
- eine ausführliche Beschreibung der Wirkungsweise.

1. Ein allererster Gehversuch mit Hypra-Ass

Zu diesem und zu allen folgenden Programmen benötigen Sie den Hypra-Ass. Er wird wie ein Basic-Programm geladen und mit RUN initialisiert. Jedes weitere RUN startet jetzt einen Assemberlauf, ist aber zunächst noch wirkungslos, da noch nichts im Textspeicher des Assemblers steht. Geben Sie nun folgende vier Zeilen ein:

```
10 - .BA $C000
20 - LDA #1
30 - STA $400
40 - RTS
```

Dabei dürfen die Minus-Zeichen nach den Zeilennummern nicht vergessen werden (Eigenart von Hypra-Ass). Auch das Leerzeichen nach den Minus-Zeichen ist wichtig. Sie werden feststellen, daß der Assember die Zeilen nach Drücken der RETURN-Taste formatiert. Listen Sie die vier Zeilen auch einmal probeweise mit

LIST (unformatiert) und
/E (formatiert)

Das Programm ist schnell erklärt:

Mit .BA $C000 wird dem Assembler die Startadresse (BAsisadresse) des Programms mitgeteilt. .BA ist ein Pseudobefehl. Ein solcher Befehl steuert die Arbeitsweise des Assemblers, bewirkt aber keine Erzeugung eines Maschinenbefehls.

LDA #1 lädt den Akkumulator mit 1, dem Bildschirmcode des Buchstabens A.

STA $400 speichert den Akkumulatorinhalt, also die 1 an die Speicherstelle $400. $400 ist die Startadresse des Bildschirm-RAMs und entspricht der linken oberen Bildschirmecke. Dort muß also ein »A« erscheinen.

RTS (ReTurn from Subroutine) bedeutet Rücksprung aus einem Unterprogramm. Mit RTS müssen Programme abgeschlossen sein, die mit JSR (Jump to SubRoutine) oder SYS (von Basic aus) aufgerufen werden. Dies dürfte für die überwiegende Mehrheit aller Maschinenprogramme der Fall sein. Ausnahmen sind:

Programme., die von einem Monitor aus gestartet werden. Sie sollten mit einem BRK (Break) abgeschlossen sein.

Programme, die durch Interrupts aktiviert werden, werden normalerweise durch RTI (ReTurn from Interrupt) abgeschlossen.

Mit RUN wird der Assembler gestartet. Er erzeugt ein 6 Byte kurzes Maschinenprogramm ab $C000 und gibt in einer Abschlußmeldung den belegten Speicherbereich zusammen mit der Assemblierzeit bekannt. Das Maschinenprogramm kann nun mit

```
SYS $C000 oder SYS 12 * 4096
```

gestartet werden. (Wenn Hypra-Ass aktiv ist, versteht der C 64 auch Hex-Zahlen). Es müßte ein »A« in der linken oberen Bildschirmecke erscheinen. Sollte das nicht der Fall sein, so kann das zwei Ursachen haben:

1. Das »A« wurde nach oben weggescrollt, weil Sie den SYS-Befehl zu weit unten auf dem Bildschrim eingetippt haben.
2. Sie besitzen eine alte Version des C 64, bei der das Farb-RAM mit der Hintergrundfarbe vorbesetzt wird.

Im zweiten Fall ergänzen Sie das Programm durch:

```
35 - STA $D800
```

Dieser Befehl speichert die immer noch im Akkumulator (im folgenden nur Akku genannt) stehende 1 an die Startadresse des Farb-RAMs. Dadurch erscheint das »A« in weißer Farbe.

Ergänzen Sie Ihr kleines Programm einmal durch den Pseudobefehl

```
5 - .LI 1,3,0
```

und assemblieren Sie mit RUN. Der Assembler erzeugt jetzt ein Listing, dessen Zeilen von links nach rechts wie folgt aufgebaut sind:
- Speicheradresse des folgenden Maschinenbefehls
- Der Code des Maschinenbefehls. ,Da es Maschinenbefehle mit ein, zwei oder drei Byte gibt, sind diese Einträge unterschiedlich lang.
- Ein Doppelpunkt und die ursprüngliche Quelltextzeile. Bei Pseudobefehlen, die ja keinen Code erzeugen, entfällt der Teil bis einschließlich zum Doppelpunkt.

Das Assemblerlisting ist bei der Fehlersuche mit einem Monitor nützlich, da es zu jedem Maschinenbefehl seine Adresse enthält. Mit dem Pseudobefehl:

```
.li 1,4,0
```

erhält man ein Druckerlisting. Angenehm dabei ist, daß es bei ausgeschaltetem Drucker automatisch auf den Bildschrim umgeleitet wird.

Die abgedruckten Listings enthalten allerdings keine Speicheradressen mit zugehörigem Maschinencode. Diese Information ist zum Studieren der Programme uninteressant und zum Eintippen der Listings nicht erforderlich. Die Listings wurden mit dem Editorbefehl /E (formatiertes Listen) gewonnen, nachdem vorher die Ausgabe durch

```
OPEN 4,4:CMD 4
```

auf den Drucker umgeleitet wurde.

2. Eine einfache Programmschleife

Das folgende kleine Programm (Listing 1) schreibt 240mal den Buchstaben »A« auf den Bildschirm. Zur Arbeitsweise: Akku A wird wieder mit dem Bildschirmcode des Buchstabens A geladen. Das X-Register übernimmt zwei Aufgaben: Es zählt Schleifendurchläufe und liefert Werte zur Adreßverschiebung. X wird mit 0 vorbesetzt. In Zeile 1280 ist

```
100  -;--------------------------------
110  -; PROGRAMMSCHLEIFE
120  -; 240 MAL "A"
130  -; AUF BILDSCHIRM SCHREIBEN
140  -;--------------------------------
150  -          .BA $C000       ;PROGRAMM-STARTADRESSE
160  -          LDA #1          ;BILDSCHIRMCODE VON "A"
170  -          LDX #0          ;SCHLEIFENZAEHLER
180  -LOOP      STA $400,X      ;ZEICHEN AUF BILDSCHIRM SCHREIBEN
190  -          INX             ;X HOCHZAEHLEN
200  -          CPX #240        ;X MIT 240 VERGLEICHEN
210  -          BNE LOOP        ;FALLS UNGLEICH, GEHE NACH "LOOP"
220  -          RTS             ;FALLS X=240, DANN PROGRAMMENDE

READY.
```

Listing 1. Programmschleife

LOOP ein Label (auch SYMBOL oder Sprungmarke). LOOP steht für die Adresse des STA-Befehls, die man an anderen Programmstellen durch den Namen LOOP ansprechen kann, ohne daß man den Wert dieser Adresse kennt. Den Assemblerprogrammierer interessieren absolute Adressen normalerweise auch gar nicht, es sei denn, es handelt sich um systemspezifische Adressen, wie zum Beispiel die Adressen der Video-Chip-Register. Eine derartige Zuordnung einer Programmadresse zu einem Label nennt man implizite Definition.

Die Zieladresse des STA-Befehls in Zeile 180 ergibt sich durch die Summe aus $400 und X. Da X am Anfang 0 gesetzt worden ist, wird also eine 1 an die Stelle $400 gespeichert. INX erhöht X um Eins. CPX #240 vergleicht X mit der Zahl 240. Bei Ungleichheit wird das Zero-Flag im Statusregister auf 0 gesetzt. Auf dieses Zero-Flag bezieht sich dann der bedingte Sprung BNE LOOP. BNE springt dann, wenn der vorige Vergleich Ungleichheit ergeben hat. (Daher auch der Name BNE = Branch if Not Equal = verzweige, wenn ungleich). Beim nächsten Schleifendurchlauf wird die 1 aus dem Akku an die Adresse $4012 gespeichert. X wird solange inkrementiert, bis 240 erreicht ist. In diesem Fall springt BNE nicht und das Programm endet mit RTS. Bei den 240 Schleifendurchläufen werden nacheinander die Adressen $400 bis $400+239 angesprochen. Programmschleifen wie diese benützt man oft zum Löschen eines Speicherbereichs. (Akku mit 0 vorbesetzt.)

```
100  -;--------------------------------
110  -; BLOCKVERSCHIEBUNG
120  -; (MAXIMAL 255 BYTE)
130  -;--------------------------------
140  -          .BA $C000       ;PROGRAMMSTART
150  -          .EQ QUELLE=$400;BLOCKSTART
160  -          .EQ ZIEL=$400+240
170  -          .EQ LAENGE=240 ;BLOCKLAENGE
180  -; X LAEUFT RUECKWAERTS VON LAENGE BIS 1
190  -          LDX #LAENGE
200  -LOOP      LDA QUELLE-1,X
210  -          STA ZIEL-1,X
220  -          DEX             ;X:=X-1
230  -          BNE LOOP        ;FALLS X<>0, NACH LOOP
240  -          RTS             ;SONST ENDE
READY.
```

Listing 2. Blockverschiebung

```
100  -;--------------------------------
110  -; BLOCKVERSCHIEBUNG
120  -; OHNE EINSCHRAENKUNGEN
130  -;--------------------------------
140  -          .BA $C000       ;PROGRAMMSTART
150  -          .EQ VON=$A09E
160  -          .EQ BIS=$A327
170  -          .EQ ZIEL=$400
180  -          .EQ ZEIGER1=$FB
190  -          .EQ ZEIGER2=$FD
200  -;
210  -          LDA #<(VON)
220  -          STA ZEIGER1
230  -          LDA #>(VON)
240  -          STA ZEIGER1+1
250  -          LDA #<(ZIEL)
260  -          STA ZEIGER2
270  -          LDA #>(ZIEL)
280  -          STA ZEIGER2+1
290  -          LDY #0
300  -LOOP      LDA (ZEIGER1),Y
310  -          STA (ZEIGER2),Y
320  -; ZEIGER1 MIT "BIS" VERGLEICHEN
330  -          LDA ZEIGER1
340  -          CMP #<(BIS)
350  -          BNE WEITER
360  -; LOW-BYTES STIMMEN UEBEREIN, HIGH-BYTES VERGLEICHEN
370  -          LDA ZEIGER1+1
380  -          CMP #>(BIS)
390  -          BEQ ENDE
400  -;         BEIDE ZEIGER INKREMENTIEREN
410  -WEITER    INC ZEIGER1
420  -          BNE WEITER2
430  -          INC ZEIGER1+1
440  -WEITER2   INC ZEIGER2
450  -          BNE LOOP
460  -          INC ZEIGER2+1
470  -          JMP LOOP
480  -ENDE      RTS

READY.
```

Listing 3. Blockverschiebung ohne Einschränkung

3. Blockverschiebung (maximal 255 Byte)

Das Programm (Listing 2) arbeitet mit einer ähnlichen Schleife wie das vorige. In den Zeilen 150, 160 und 170 werden Label explizit definiert. Dies geschieht mit dem Pseudobefehl .EQ.

Die explizite Definition eines Labels ist praktisch dasselbe wie die Zuweisung eines Wertes an eine Variable. Hier werden die Anfangsadressen des ursprünglichen Blocks und des verschobenen Blocks sowie die Blocklänge definiert. X läuft hier rückwärts von LAENGE bis 0. Dadurch kann der CPX-Befehl eingespart werden. DEX setzt nämlich automatisch das Zero-Flag, wenn nach dem Dekrement X den Wert 0 hat. BNE LOOP springt also nur solange nach LOOP, solange X größer als 0 ist. Mit den vorliegenden Werten für QUELLE, ZIEL und LAENGE kopiert das Programm die Bildschirmzeilen 1 bis 6 auf die Zeilen 7 bis 12.

Programme zur Blockverschiebung wie dieses oder zur Blockfüllung wie das vorige sind nur für Blocklängen bis maximal 255 Byte geeignet, da das X-Register nur 8 Bit lang ist. Wenn man größere Speicherbereiche auf diese Weise verarbeiten will, muß man mehr Aufwand treiben.

4. Blockverschiebung (ohne Einschränkungen)

Das Programm (Listing 3) ist sicher nicht die kürzeste Lösung des Problems, es demonstriert dafür aber ohne verwirrende Tricks die Adressierungsart »Indirekt Indiziert«.

Beispiel: `LDA (ZEIGER),Y`

Bei dieser Adressierungsart enthalten zwei aufeinanderfolgende Speicherstellen der Zero-Page eine Adresse in der üblichen Reihenfolge Low-Byte - High-Byte. Im Programm wird nicht diese Adresse selbst angegeben, sondern die Adresse der ersten der beiden Zero-Page-Speicherstellen (hier ZEIGER genannt). Diese Technik nennt man indirekte Adressierung, was im Assemblertext durch die runden Klammern um die Zero-Page-Adresse zum Ausdruck kommt. Zu der aus der Zero-Page stammenden Adresse wird noch Y addiert, daher »indiziert«. Da man diese zusätzliche Indizierung oft nicht braucht, setzt man das Y-Register vorher auf O.

Das Programm verwendet für den Blocktransfer zwei Zeiger (= Zero-Page-Speicherstellenpaare). Sie werden mit der Startadresse des Quell- beziehungsweise des Zielblocks initialisiert und nach jedem Byte-Transfer hochgezählt, bis der Zeiger in den Quellblock (ZEIGER1) das Ende des Quellblocks (Adresse BIS) erreicht hat.

Das Inkrementieren eines 16-Bit Wertes verläuft nach dem Schema:

```
      INC ZEIGER  ;Low-Byte inkrementieren
      BNE Weiter  ;falls ungleich 0, dann fertig
      INC ZEIGER+1;Übertrag ins High-Byte
WEITER (Programmfortsetzung)
```

Spezifisch für den Hypra-Ass ist, daß man mit
<(Adresse) beziehungsweise >(Adresse)
das Low- bezeihungsweise High-Byte einer Adresse (beziehungsweise eines Labels) gezielt ansprechen kann. Von dieser Möglichkeit wird im Programm häufig Gebrauch gemacht. So bedeutet zum Beispiel:

```
LDA #<     (VON)
```

Lade den Akkumulator mit dem Low-Byte des Wertes VON.

Mit den im Program definierten Adressen VON, BIS und ZIEL kopiert das Programm einen Teilbereich aus dem Basic-

Interpreter direkt auf den Bildschrim. Im Groß-/Kleinschriftmodus (Commodore-Shift drücken) kann man dann Basic-Schlüsselwörter sowie Texte von Fehlermeldungen lesen.

5. Verwendung von Betriebssystem-Funktionen und Mechanismen zur Parameterübergabe

Ein Betriebssystem ist unter anderem dazu da, Standarddienste wie Ein- und Ausgabe zur Verfügung zu stellen, damit diese nicht jedesmal mühsam und fehleranfällig neu programmiert werden müssen. Die Standardfunktionen des Betriebssystems (oft »Kernel« genannt) sind im Programmierhandbuch von Commodore hinreichend erläutert. Viele weitere nützliche Routinen findet man beim Studium eines kommentierten ROM-Listings.

Die Parameterübergabe an Maschinensprache-Unterprogramme gestaltet sich leider nicht so systematisch wie bei den meisten höheren Programmiersprachen. Es werden mehrere Möglichkeiten bunt gemischt angewendet.

1. Man schreibt Parameter in vereinbarte Speicherstellen. Aus diesen holt sich dann das aufgerufene Programm die Parameter.
2. Wenn nur Ein- bis Drei-Byte-Parameter benötigt werden, kann man diese auch in den Registern A, X und Y übergeben. Auf diese Weise werden die meisten Kernel-Funktionen mit Parametern versorgt.
   Dieser Mechanismus steht übrigens auch von Basic aus zur Verfügung: Man schreibt Registerparameter per POKE an speziell dafür vorgesehene Speicherstellen:

Akku A	780 ($30C)
Index X	781 ($30D)
Index Y	782 ($30E)
Status-Register	783 ($30f)

   Das Maschinenprogramm (Listing 4) wird nun mit SYS aufgerufen. Der Basic-Interpreter besetzt erst die Register mit den Inhalten dieser Speicherstellen und bringt dann in das Unterprogramm. Nach der Rückkehr werden die (neuen) Registerinhalte wieder in denselben Speicherstellen abgelegt, wo sie für eine eventuelle Inspektion durch das Basic-Programm zur Verfügung stehen.

```
100  -;--------------------------------
110  -; VERWENDUNG DER BETRIEBSSYSTEM-
120  -; ROUTINEN GETIN UND CHROUT
130  -;--------------------------------
140  -          .BA $C000
150  -          .EQ GETIN=$FFE4
160  -          .EQ CHROUT=$FFD2
170  -          .EQ ZAEHLER=$FE
180  -          .EQ MAXLEN=10  ;ZEILENLAENGE
190  -          .EQ PROMPT=63  ;"?"
200  -          .EQ ESCAPE=88  ;FLUCHTSYMBOL "X"
210  -          .EQ SPACE=32
220  -          .EQ CR=13      ;CARRIAGE RETURN
230  -;
240  -NEWLINE   LDA #MAXLEN    ;ZAEHLER INITIALISIEREN
250  -          STA ZAEHLER
260  -          LDA #CR
270  -          JSR CHROUT     ;ZEILENVORSCHUB
280  -          LDA #PROMPT
290  -          JSR CHROUT     ;PROMPT-ZEICHEN AUSGEBEN
300  -          LDA #SPACE
310  -          JSR CHROUT
320  -WAIT      JSR GETIN      ;AUF EINGABE WARTEN
330  -          CMP #0
340  -          BEQ WAIT
350  -          CMP #ESCAPE    ;BEI ESCAPE-ZEICHEN PROGRAMMENDE
360  -          BEQ ENDE
370  -          CMP #CR        ;BEI CR NEUE ZEILE
380  -          BEQ NEWLINE
390  -          JSR CHROUT     ;EINGABEZEICHEN WIEDER AUSGEBEN
400  -          DEC ZAEHLER
410  -          BNE WAIT       ;NAECHSTE EINGABE
420  -          BEQ NEWLINE    ;ZEILENLAENGE ERREICHT
430  -ENDE      RTS

READY.
```

Listing 4. Verwendung von Betriebssystemroutinen

3. Man kann Parameter auch über den Stack übergeben. Diese Methode ist wegen des kleinen Stackbereichs der 6510-CPU (256 Byte) nur bedingt brauchbar und wird deshalb auch kaum praktiziert.
4. Durch geschickte Verwendung von Unterprogrammen in Basic-ROM kann man Parameter direkt hinter den SYS-Befehl schreiben. Diese Methode ist
   - komfortabel, weil keine umständlichen POKEs nötig sind
   - schnell, weil der Interpreter weniger zu tun hat
   - flexibel, weil als Parameter auch ganze arithmetische oder Stringausdrücke geschrieben werden können.

   Diese Methode wird in den Programm-Listings 5 bis 8 verwendet.

Das folgende Programm (Listing 4) nutzt die Funktionen GETIN und CHROUT. GETIN liefert den ASCII-Code einer gedrückten Taste im Akku. Falls keine Taste gedrückt wurde, wird 0 zurückgegeben. GETIN entspricht damit genau dem GET-Befehl in Basic.

CHROUT gibt ein Zeichen, dessen ASCII-Code im Akku stehen muß, auf dem Bildschrim aus. Es entspricht dem Basic-Befehl (man beachte das Semikolon):

```
PRINT CHR$(A);
```

Das Programm gibt einen Prompt aus und erwartet anschließend Eingaben. Unter einem Prompt versteht man ein (beliebig zu vereinbarendes) Zeichen am linken Bildschirmrand, das dem Benutzer mitteilt, daß Eingaben von ihm erwartet werden. Bei interaktiven Programmen (wie zum Beispiel Monitore, Editore) sind Prompts sehr nützlich, da der Benutzer daran eindeutig erkennen kann, in welchem Programm er gerade ist. Das vorliegende Programm gibt die Eingabezeichen sofort wieder aus, ohne sie weiter zu verarbeiten. Nach maximal zehn Zeichen wird automatisch ein Zeilenvorschub ausgeführt und ein weiterer Prompt ausgegeben. Das Programm ist eine Endlosschleife, die man mit der Eingabe von »X« verlassen kann.

6. Verwendung von Interpreter-Routinen zur Parameterübergabe

Diese Interpreter-Routinen werden in den folgenden Programmbeispielen eingesetzt:

CHKKOM ($AEFD)	liest aus dem laufenden Basic-Text ein Komma. Steht an der aktuellen Stelle kein Komma, wird das Programm mit SYNTAX ERROR abgebrochen. Kommata sind nötig, um Parameter voneinander abzugrenzen.
FRMNUM ($AD8A)	wertet einen beliebigen arithmetischen Ausdruck aus. Das Ergebnis wird im Fließkomma-Akkumulator 1 (kurz FAC) abgelegt. Der FAC besteht aus den Speicherstellen $61-$66. Die Bedeutung der einzelnen Byte ist hier nicht relevant.
GETADR ($B7F7)	wandelt den Inhalt des FAC in ein 2-Byte-Integer-Format um, sofern diese Zahl im Bereich 0 ... 65535 liegt. Ansonsten wird ein ILLEGAL QUANTITY ERROR ausgegeben. Die Integerzahl steht in den Speicherstellen $14/$15 und zusätzlich im Registerpaar Y/A. Mit der Kombination FRMNUM und GETADR kann man also 16-Bit-Größen aus Basic-Programmen übernehmen.
XBYTE ($B79E)	wertet ebenfalls arithmetische Ausdrücke aus und wandelt das Ergebnis in 8-Bit-Integerformat, sofern es im Bereich 0 ... 255 liegt. Das Byte-Ergebnis wird im X-Register übergeben.
USR ( )	ist eine Basic-Funktion, mit der man Werte von Maschinenprogrammen an Basic zurückgeben kann. USR wertet einen in Klammern stehenden

Ausdruck aus und übergibt ihn in den FAC. Es wird ein Maschinenprogramm aufgerufen, dessen Startadresse in $311/$312 steht. (USR-Vektor). Das Maschinenprogramm kann dann im FAC einen Wert an Basic zurückgeben.

Das folgende Listing (Listing 5) ist der Programmkopf zu den vier nachfolgenden Beispielen. Diese können mit dem Kopf zusammen assembliert werden. Der Kopf enthält eine Sprungliste. Dadurch werden Einsprungstellen ($C000, $C003, etc.) für die vier aufgeführten Programme fixiert, unabhängig davon, wo die Programme dann später tatsächlich im Speicher stehen. Diese Technik ist zum Beispiel

```
100  -;================================
110  -; EINIGE ALLGEMEIN NUETZLICHE
120  -; MASCHINENSPRACHE-UNTERPROGRAMME
130  -; FUER DEN AUFRUF DURCH
140  -; BASIC-PROGRAMME
150  -;
160  -; EINFACHE PARAMETERUEBERGABE:
170  -; SYS STARTADRESSE,PARAMETERLISTE
180  -;================================
190  -;
200  -          .BA $C000
210  -; EINSPRUNGPUNKTE UND UNTERROUTINEN
220  -; DES BASIC-INTERPRETERS
230  -          .EQ CHKKOM=$AEFD;PRUEFT AUF KOMMA
240  -          .EQ FRMNUM=$AD8A;BERECHNET NUMERISCHEN AUSDRUCK IN FAC
250  -          .EQ GETADR=$B7F7;WANDELT FAC IN INTEGERFORMAT ($14/$15)
260  -          .EQ XBYTE=$B79E;HOLT BYTE-WERT NACH X
270  -          .EQ PLOT=$FFF0 ;CURSOR SETZEN
280  -          .EQ PRINT=$AAA0;BASIC-PRINT
290  -          .EQ SETLFS=$FFBA;FILEPARAMETER SETZEN
300  -          .EQ SAVE=$FFD8
310  -;
320  -; SPRUNGLISTE
330  -;
340  -          JMP PRINTAT
350  -          JMP DEEK
360  -          JMP DOKE
370  -          JMP SAV

READY.
```

Listing 5. Einfache Parameterübergabe aus Basic-Programmen

sinnvoll, wenn mehrere Leute zusammen an einem größeren Programm arbeiten. Ein Programmierer kann seinen Kollegen bereits feste Einsprungstellen für Routinen, an denen er noch arbeitet oder die noch gar nicht existieren, zur Verfügung stellen.

PRINT AT

Das Programm (Listing 6) ermöglicht eine freie und schnelle Cursorpositionierung zusammen mit einer Druckausgabe. Mit der Definition

```
PR=123*4096 :REM Startadresse
```

kann mit

```
SYSPR,Zeile,Spalte,Printliste
```

alles ausgeben werden, was auch mit PRINT ausgegeben

```
380  -;
390  -;--------------------------------
400  -; PRINT AT
410  -; AUFRUF: SYSPR,ZEILE,SPALTE,PRINTLISTE
420  -;--------------------------------
430  -PRINTAT   JSR CHKKOM    ;1. KOMMA
440  -          JSR XBYTE     ;ZEILE NACH X
450  -          TXA
460  -          PHA           ;AUF STACK ZWISCHENSPEICHERN
470  -          JSR CHKKOM    ;2. KOMMA
480  -          JSR XBYTE     ;SPALTE NACH X
490  -          TXA
500  -          TAY           ;SPALTE NACH Y
510  -          PLA
520  -          TAX           ;ZEILE NACH X
530  -          CLC
540  -          JSR PLOT      ;CURSORPOSITION SETZEN
550  -          JSR CHKKOM    ;3.KOMMA
560  -          JMP PRINT     ;WEITER MIT BASIC-PRINT

READY.
```

Listing 6. PRINT AT-Befehl selbstgemacht

```
570  -;
580  -;--------------------------------
590  -; DEEK (16-BIT-PEEK)
600  -; DER USR VEKTOR ($311/$312)
610  -; MUSS AUF DIESES PROGRAMM ZEIGEN
620  -; AUFRUF: USR(ADRESSE)
630  -;--------------------------------
640  -DEEK      JSR GETADR     ;FAC NACH INTEGER ($14/15)
650  -          LDY #0
660  -          SEI
670  -          LDA ($14),Y    ;LOW-BYTE
680  -          STA $63        ;FAC MANTISSE
690  -          INY
700  -          LDA ($14),Y    ;HIGH-BYTE
710  -          CLI
720  -          STA $62        ;FAC MANTISSE
730  -          LDX #$90       ;FAC EXPONENT
740  -          SEC            ;NICHT INVERTIEREN
750  -          JMP $BC49      ;FAC KOMPLETT MACHEN
```

Listing 7. Eigener DEEK-Befehl

werden kann. Man lasse sich einmal von der Geschwindigkeit des folgenden Programms beeindrucken:

```
10 FOR I=1 TO 24:SYSPR, I,I,"A":NEXT
20 FOR I=1 TO 24:SYSPR,-I,I,"B":NEXT
```

Das Assemberlisting zu PRINT AT bedarf keiner großen Erläuterung. PLOT ist eine Kernel-Funktion, mit der man die Cursorposition auf dem Bildschirm setzen kann. Parameter sind Zeilen- und Spaltennnummern in den Registern X und Y. Das Programm PRINT AT ist eigentlich nicht mehr als eine geschickte Kombination der Routinen PLOT und PRINT.

DEEK (Doppelbyte-PEEK)

Dieses Programm (Listing 7) ist eine Abänderung der PEEK-Routine. DEEK liefert einen 16-Bit-Speicherinhalt an Basic zurück. DEEK wird durch

```
X=USR(Adresse)
```

aufgerufen. Mit Adresse ist die Adresse des Low-Bytes gemeint. Da USR einen Wert zurückgibt, darf es nicht isoliert dastehen, sondern muß als rechte Seite einer Zuweisung oder als Funktionsargument eingesetzt werden. Vor dem ersten Aufruf muß der USR-Vektor auf die Startadresse des Programms gestellt werden:

```
POKE 785,3   :REM LOW-BYTE  $03
POKE 786,192 :REM HIGH-BYTE $C0
```

Im Assemblerlisting steckt eine Besonderheit: Die Zugriffe auf die beiden zu lesenden Bytes (LDA ($14),Y) sind durch ein SEI/CLI-Paar eingerahmt. SEI sperrt die CPU für Interruptanforderungen. Dadurch wird garantiert, daß die beiden Lesezugriffe nicht durch ein Interruptprogramm, welches eines oder beide Bytes ändern könnte, unterbrochen werden können. CLI löst die Interruptsperre wieder.

DOKE (Doppelbyte-POKE)

Um in Basic-Programmen 16-Bit-Größen (zum Beispiel Adressen, Vektoren) in den Speicher zu schreiben, muß man sie vorher erst umständlich in High- und Low-Byte zerlegen, um dann beide Byte POKEn zu könen. Dazu wird meistens die Sequenz:

```
HI=INT(X/256)
LO=X-256*HI
POKE AD,LO
POKE AD+1,HI
```

verwendet.

Wenn man bedenkt, daß jeder Befehl interpretiert werden muß und daß jede Rechenoperation (auch »+1«) in voller Fließkomma-Genauigkeit durchgeführt wird, versteht man, daß dazu viel Rechenzeit nötig ist. Das kleine Maschinenprogramm (Listing 8), das keiner Erläuterung mehr bedarf (FRMNUM, GETADR und CHKKOM sind bekannt) zeigt, wie es einfacher geht:

```
SYSDO,AD,Y
```

```
760  -;
770  -;--------------------------------
780  -; DOKE (16-BIT-POKE)
790  -; AUFRUF: SYSDO,ADRESSE,WERT
800  -;--------------------------------
810  -DOKE      JSR CHKKOM      ;1. KOMMA
820  -          JSR FRMNUM      ;ADRESSE NACH FAC
830  -          JSR GETADR      ;FAC NACH INTEGER ($14/15)
840  -          LDA $14
850  -          STA $9E         ;ADRESSE NACH $9E/9F
860  -          LDA $15
870  -          STA $9F
880  -          JSR CHKKOM      ;2.KOMMA
890  -          JSR FRMNUM      ;WERT NACH FAC
900  -          JSR GETADR      ;FAC NACH INTEGER ($14/15)
910  -          LDY #0
920  -          SEI
930  -          LDA $14         ;WERT LOW-BYTE
940  -          STA ($9E),Y
950  -          INY
960  -          LDA $15         ;WERT HIGH-BYTE
970  -          STA ($9E),Y
980  -          CLI
990  -          RTS

READY.
```

Listing 8. Eigener DEEK-Befehl

(Natürlich muß man DO einmal vorher definieren: DO=12*4096+6). Auch bei DOKE werden die beiden kritischen STA-Befehle durch ein SEI/CLI-Paar untrennbar gemacht. Mit DOKE kann man daher sogar den Interrupt-Vektor ändern. Versucht man dies dagegen mit Hilfe zweier POKEs, kann es passieren, daß ein Interrupt gerade dann auftritt, nachdem das Low-Byte aber noch nicht das High-Byte geändert worden ist. Der Interrupt führt dann auf eine unbestimmte Adresse, was meistens einen Programmabsturz nach sich zieht.

DEEK und DOKE können natürlich auch verschachtelt eingesetz werden. So kann man mit

```
SYSDO,A2,USR(A1)
```

einen 16-Bit-Wert von der Stelle A1 nach A2 kopieren.

Speichern beliebiger Speicherbereiche auf Diskette

Das Programm (Listing 9) realisiert das Gegenstück zum Basic-Befehl:

```
LOAD "Name",8,1
```

```
1000 -;
1010 -;--------------------------------
1020 -; SAV
1030 -; SPEICHERE BELIEBIGEN BEREICH AUF DISK
1040 -; AUFRUF: SYSSAV,DATEINAME,GERAETENUMMER,STARTADRESSE,ENDADRESSE
1050 -;--------------------------------
1060 -SAV       JSR CHKKOM      ;1. KOMMA
1070 -          JSR $E257       ;FILENAMEN HOLEN UND SETZEN
1080 -          JSR CHKKOM      ;2. KOMMA
1090 -          JSR XBYTE       ;GERAETENUMMER NACH X
1100 -          LDY #0          ;SEKUNDAERADRESSE
1110 -          JSR SETLFS      ;FILEPARAMETER SETZEN
1120 -          JSR CHKKOM      ;3. KOMMA
1130 -          JSR FRMNUM      ;STARTADRESSE
1140 -          JSR GETADR      ;NACH $14/15 UND Y/A
1150 -          PHA             ;HIGH-BYTE
1160 -          TYA
1170 -          PHA             ;LOW-BYTE
1180 -          JSR CHKKOM      ;4. KOMMA
1190 -          JSR FRMNUM      ;ENDADRESSE
1200 -          JSR GETADR      ;NACH $14/15 UND Y/A
1210 -          PHA
1220 -          TYA
1230 -          TAX
1240 -          PLA
1250 -          TAY             ;ENDADRESSE LOW IN X, HIGH IN Y
1260 -          PLA             ;STARTADRESSE LOW-BYTE
1270 -          STA $14
1280 -          PLA             ;STARTADRESSE HIGH-BYTE
1290 -          STA $15         ;STARTADRESSE IN $14/15
1300 -          LDA #$14        ;ADRESSE DER STARTADRESSE
1310 -          JSR SAVE
1320 -          BCC SAVENDE     ;KEIN FEHLER
1330 -          JMP $E0F9       ;FEHLERAUSGANG
1340 -SAVENDE   RTS

READY.
```

Listing 9. Speichern von beliebigen Speicherbereichen

Aufgerufen wird es durch:

```
SYSSAV,Dateiname,gn,sa,ea
```

Dabei kann bei »Dateinamen« ein Name oder ein Stringausdruck in Anführungszeichen stehen.
»gn« ist die Gerätenummer (8 oder 9)
»sa« und »ea« sind Start- und Endadresse des abzuspeichernden Bereiches. Zum Programm selbst:

Die Routine bei $E257 beschafft sich den Filenamen aus dem Basic-Text und stellt ihn der später folgenden SAVE-Routine zur Verfügung. Mit SETLFS kann man dem Betriebssystem eine logische Filenummer (im Akku), eine Gerätenummer (in X) und eine Sekundäradresse (in Y) bekanntgeben. Die Parametrisierung der Kernel-SAVE-Routine ist etwas komplizierter:

X Endadresse Low-Byte
Y Endadresse High-Byte
A Zeiger auf das untere Byte eines Zero-Page-Bytepaares, welches die Startadresse enthält.

Die SAVE-Routione kehrt mit gesetztem Carry-Flag zurück, falls beim Speichern ein Fehler aufgetreten ist. Das Programm bei $E0F9 sorgt dann für eine ordentliche Fehlermeldung.

```
100  -;--------------------------------
110  -; MULTIPLIKATION  8 MAL 8 BIT
120  -;
130  -;  MD MULTIPLIKAND (BLEIBT ERHALTEN)
140  -;  MR MULTIPLIKATOR (WIRD UEBERSCHRIEBEN)
150  -;
160  -; DAS 16-BIT-PRODUKT STEHT IN:
170  -;  MR (HIGH-BYTE) UND
180  -;  A  (LOW-BYTE)
190  -;--------------------------------
200  -          .BA $C000
210  -          .EQ MD=$FD
220  -          .EQ MR=$FE
230  -MUL       LDA #0          ;VORBESETZUNG DES PRODUKTS
240  -          LDX #8          ;ZAEHLER (8 DURCHLAEUFE)
250  -MULLOOP   ASL             ;PRODUKT IN A UEBER
260  -          ROL MR          ;MR NACH LINKS SCHIEBEN
270  -          BCC MULNEXT     ;HOECHSTES BIT IN MR=0
280  -          CLC             ;FALLS HOECHSTES BIT IN MR=1,
290  -          ADC MD          ;MD ZUM TEILPRODUKT ADDIEREN
300  -          BCC MULNEXT     ;KEIN UEBERTRAG
310  -          INC MR          ;UEBERTRAG NACH MR BERUECKSICHTIGEN
320  -MULNEXT   DEX
330  -          BNE MULLOOP     ;WEITER, FALLS ZAEHLER NOCH NICHT 0
340  -          RTS

READY.
```

Listing 10. Mulitplikation 8 mal 8 Bit

Multiplikation

Das Programm (Listing 10) multipliziert zwei Byte-Werte miteinander und liefert ein 16-Bit-Produkt. Es ist aber trotz seiner Kürze nicht ganz einfach zu verstehen. Die beiden zu multiplizierenden Faktoren seien mit

```
Multiplikator MR und Mulitplikand MD
```

bezeichnet. Für das Resultat ist es natürlich gleichgültig, welcher Faktor als MR und welcher als MD an das Programm übergeben wird. MR kann man sich, wie jede binäre Größe, folgendermaßen vorstellen:

MR=MR(7)*128+MR(6)*64+...+MR(1)*2+MR(0)*1

Dabei bezeichnet zum Beispiel MR(6) das Bit Nummer 6 von MR in der üblichen Zählweise von 0 bis 7 und von rechts nach links. Das Produkt MR*MD kann man nun so berechnen:

Addiere folgende Teilprodukte:

```
MD*128, falls MR(7)=1, sonst 0
MD*64,  falls MR(6)=1, sonst 0
MD*2,   falls MR(1)=1, sonst 0
MD,     falls MR(0)=1, sonst 0
```

Die Teilprodukte erhält man einfach durch Linksverschieben von MD:

```
MD*128 durch 7-maligen Links-Shift
MD*64  durch 6-maligen Links-Shift
```

Zu addieren ist nur dann etwas, wenn das entsprechende Bit in MR=1 ist. Wenn man MR mit dem ROL-Befehl achtmal

nach links schiebt, so durchwandern alle 8 Bit nacheinander das Carry-Flag und letzteres kann leicht abgefragt werden. Der Trick des Programms besteht nun darin, daß das Berechnen von Teilproduktsummen mit dem Linksschieben von MR kombiniert wird. Zunächst wird die Zwischensumme in A mit 0 vorbesetzt. MR wird nach links geschoben. Das höchstwertige Bit von MR steht jetzt im Carry-Flag. Ist es 1, so wird MD zum Akku addiert. Eigentlich müßte jetzt der Akku um sieben Positionen nach links geschoben werden, da zum ersten Teilprodukt der Faktor 128 gehört. Diese Verschiebung ergibt sich aber automatisch im Verlauf der nächsten sieben Schleifendurchläufe.

Sowie MR nach links geschoben wird, werden rechts in MR Bits frei, die dann von den von rechts kommenden höherwertigen Bits der Zwischensumme belegt werden. Nach 8 Schleifendurchläufen ist schließlich MR nach links verdrängt worden. An seiner Stelle steht nun das High-Byte des Produkts. Das Low-Byte des Produkts steht im Akku, während MD unverändert geblieben ist.

```
100  -;--------------------------------
110  -; SCHIEBEREGISTER-FOLGEN
120  -; ALS PSEUDO-ZUFALLSZAHLEN
130  -;--------------------------------
140  -          .BA $C000
150  -          .EQ SR=$FD     ;SCHIEBEREGISTER (2 BYTE)
160  -          .EQ ZAEHLER=$FB;(2 BYTE)
170  -          .EQ DELAY=$FA  ;PAUSENLAENGE
180  -;
190  -          JMP MAIN       ;ZUM HAUPTPROGRAMM
200  -;
210  -; NAECHSTE ZUFALLSZAHL
220  -;  (1) CARRY := SR(6) EOR SR(9)
230  -;  (2) SR NACH LINKS SCHIEBEN
240  -;  (3) SR(0) := CARRY
250  -;
260  -SHIFT     LDA SR+1
270  -          AND #2         ;SR(9) ISOLIEREN
280  -          ASL
290  -          ASL
300  -          ASL            ;IN BITPOSITION 6
310  -          ASL            ;BRINGEN
320  -          ASL
330  -          EOR SR         ;LIEFERT SR(6) EOR SR(9)
340  -          ASL
350  -          ASL            ;RESULTAT INS CARRY-FLAG (SCHRITT (1))
360  -          ROL SR         ;SCHRITT (2) UND (3)
370  -          ROL SR+1
380  -          RTS
390  -;
400  -; VERZOEGERUNGSSCHLEIFE
410  -;
420  -PAUSE     LDX DELAY
430  -PAUSE1    DEX
440  -          BNE PAUSE1
450  -          RTS
460  -;
470  -; HAUPTPROGRAMM
480  -; SCHIEBEREGISTER-FOLGE DER LAENGE 1023 ERZEUGEN
490  -; BILDSCHIRMZEICHEN IN DER REIHENFOLGE DIESER ZUFALLSZAHLEN
500  -; INVERTIEREN (D.H. BIT 7 INVERTIEREN)
510  -;
520  -; SR MIT ZUFAELLIGEM STARTWERT VORBESETZEN
530  -MAIN      LDA $DC04      ;CIA#1 TIMER A, LOW-BYTE
540  -          ORA #1         ;DARF NICHT 0 SEIN
550  -          STA SR
560  -          LDA #$FF
570  -          STA ZAEHLER
580  -          LDA #$03
590  -          STA ZAEHLER+1  ;ZAEHLER=$3FF=1023
600  -LOOP      JSR SHIFT      ;NAECHSTE ZUFALLSZAHL
610  -          LDA SR+1
620  -          PHA            ;MERKEN
630  -          AND #3         ;HIGH-BYTE AUF 2 BIT BEGRENZEN
640  -          ORA #4         ;SR=SR+$400
650  -          STA SR+1
660  -          LDY #0
670  -          LDA (SR),Y     ;ZEICHEN VOM BILDSCHIRM
680  -          EOR #$80       ;BIT 7 INVERTIEREN
690  -          STA (SR),Y     ;ZURUECK ZUM BILDSCHIRM
700  -          PLA            ;SR+1
710  -          STA SR+1       ;WIEDERHERSTELLEN
720  -          JSR PAUSE
730  -          DEC ZAEHLER
740  -          BNE LOOP
750  -          DEC ZAEHLER+1
760  -          BPL LOOP
770  -          LDA $400       ;ERSTES BILDSCHIRMZEICHEN
780  -          EOR #$80       ;INVERTIEREN
790  -          STA $400
800  -          RTS
```

Listing 11. Schieberegister-Folgen als Pseudo-Zufallszahlen

Erwähnenswert sind hier noch die Befehle ASL und ROL: Beide schieben nach links und bei beiden wird Bit 7 ins Carry-Flag geschoben. Der Unterschied:

ASL besetzt Bit 0 mit 0

ROL besetzt Bit 0 mit dem alten Inhalt des Carry-Flags.

Mit ASL (ohne Adreßteil) wird also der Akku arithmetisch verdoppelt, während mit ROL MR zusätzlich der Übertrag aus dieser Verdoppelung in Bit 0 von MR gelangt.

Schieberegister-Folgen als Pseudo-Zufallszahlen

Das Programm (Listing 11) zeigt eine interessante Anwendung von Schieberegistern. Wenn man ein Schieberegister (SR) an den »richtigen« Bitpositionen »anzapft« und das Exklusiv-Oder-Produkt dieser Bits an den SR-Eingang zurückführt, erhält man eine Folge von Bits, die vollkommen zufällig zu sein scheint. Die Folgen sind zwar periodisch, sie wiederholen sich also nach einer gewissen Zeit, die Periodenlänge kann aber beliebig lang gemacht werden. Macht man eine so erzeugte 0-1-Folge mit einem Lautsprecher hörbar, so klingt diese wie weißes Rauschen.

Die folgende Tabelle enthält geeignete Anzapfstellen für Schieberegister unterschiedlicher Länge.

Registerlänge	Rückkopplung	Periodenlänge
2	0+1	3
3	1+2	7
4	2+3	15
5	2+4	31
6	4+5	63
7	5+6	127
8	1+2+3+7	255
9	4+8	511
10	6+9	1023
11	8+10	2047
12	1+9+10+11	4095
13	0+10+11+12	8191
14	1+11+12+13	16383
15	13+14	32767
16	10+12+13+15	65535

»+« steht hier für »EOR«

Die angegebenen Periodenlängen sind die bei der jeweiligen Registerlänge maximal möglichen. Die Schieberegisterfolgen haben die angenehme Eigenschaft, daß die Registerwerte alle Zahlen von 1 bis zur Periodenlänge in quasizufälliger Reihenfolge durchlaufen. Man darf ein solches Schieberegister allerdings nicht mit lauter Nullen vorbesetzen, da es dann seinen Zustand nicht mehr ändert (0 EOR 0 =0).

Der Kern des folgenden Programms ist die kleine Routine SHIFT. Die beiden Zero-Page-Speicherstellen SR und SR+1 bilden ein 16-Bit-Schieberegister. Rückgekoppelt wird es an den Positionen 6 und 9. Es werden quasi nur 10 Bit von den 16 vorhandenen ausgenutzt. SHIFT erzeugt bei wiederholtem Aufruf eine Folge mit der Periode 1023.

Das Hauptprogramm wendet nun diese Folge in grafisch reizvoller Weise an. Zunächst wird das Low-Byte des Schieberegisters mit einem zufälligen Wert (ungleich 0) vorbesetzt. Dieser Wert stammt aus dem ständig laufenden Timer A in CIA Nummer 1. SHIFT wird nun 1023 mal aufgerufen und erzeugt dadurch alle Zahlen von 1 bis 1023 in quasizufälliger Reihenfolge. Diese Zahlen werden als Adressen relativ zum Bildschirm-RAM verwendet. Bei den adressierten Bytes wird jeweils Bit 7 invertiert, was eine Reversdarstellung der Bildschirmzeichen bewirkt. Das Programm MAIN bewirkt also nichts anderes als eine Invertierung des gesamten Textbildschirms. Da dies aber in zufälliger Abfolge geschieht, ist der Effekt sehr auffallend. Über die Variable DELAY ($FA)=250) kann man das Tempo der Invertierung beeinflussen. Das anschließende Basic-Programm (Listing 12) erzeugt einen Flimmereffekt, indem es eine einfache Zufallsgrafik mit dem Programm MAIN invertiert. (Thomas Krätzig/aw)

```
10 POKE 250,1
20 PRINT CHR$(147)
30 FOR I=1024 TO 2024
40 POKE I,127+INT(RND(0)*2)*128
50 NEXT I
60 SYS 12*4096
70 GET A$:IF A$="" THEN 60
READY.
```

Listing 12. Basic-Hilfsprogramm zu Listing 11.