

# Maschinen- sprache mit dem C 16

**Möchten Sie Maschinenprogramme für den C16 schreiben? Wenn ja, ist dieser Artikel genau das richtige. Wo Sie welche Betriebssystem-Einsprungsadressen finden und wie Sie sie anwenden, wird hier ausführlich beschrieben.**

In diesem Artikel soll besprochen werden, welche Besonderheiten bei der Programmierung des C16 in Maschinensprache zu beachten sind, welche Zeropage-Adressen unbenutzt sind und zum Beispiel für Pointer (indirekte Adressierung) verwendet werden können, wie die Betriebssystemroutinen benutzt werden und was bei der Einbindung von Maschinenroutinen in Basic-Programme beachtet werden muß.

Alle Beispielprogramme wurden möglichst einfach gehalten und unter Verzicht auf jegliche »Tricks« erstellt. Dieser Artikel kann jedoch keinesfalls ein Maschinensprache-Lehrgang sein oder einen solchen ersetzen. Ich muß im folgenden davon ausgehen, daß der Leser Grundkenntnisse über den Prozessor des C16, dessen Befehle und die verschiedenen Adressierungsarten besitzt.

Nehmen wir an, daß Sie Assemblergrundkenntnisse besitzen oder sich vor dem Weiterlesen zuerst auf den Kurs »Assembler ist keine Alchimie« (64'er Sonderheft 8/85 »Assembler«) stürzen, und fangen an.

## Benutzung des integrierten Monitors:

Die in diesem Artikel beschriebenen Programmbeispiele sollten der Einfachheit halber mit dem eingebauten Maschinensprache-Monitor des C16 eingegeben werden. Da im Bedienungshandbuch leider jede Beschreibung des Monitors fehlt, erläutere ich kurz die wichtigsten Befehle (alle Zahlen müssen im Hexadezimalsystem eingegeben werden!):

MONITOR = Start des Monitors

X = Verlassen des Monitors

S"(FILE)",(GERÄT),(START),(ENDE+1) = Speichern eines Speicherbereichs (Programms), zum Beispiel: S"TEST",1,1000,2000 (Speichern von \$1000 bis \$1FFF unter dem Namen »TEST« auf Datasette) oder: S"TEST",8,2000,2400 (Speichern von \$2000 bis \$23FF auf Diskette).

L"(FILE)",(GERÄT) = Laden eines Programms, zum Beispiel: L"TEST",1 oder: L"TEST",8

(ADRESSE) (MNEMONIC) = Assemblieren, zum Beispiel: .1000 LDA # \$1F

> (ADRESSE) (BYTES) = Bytes eingeben, zum Beispiel: >1000 1F 10 2B

D (ANFANG) (ENDE) = Speicherbereich disassemblieren, zum Beispiel D 1000 2000

M (ANFANG) (ENDE) = Speicherbereich anschauen, zum Beispiel M 1000 2000

T (ANFANG) (ENDE) (ADRESSE) = Speicherbereich zu neuer Adresse verschieben, zum Beispiel: T 1000 2000 3000

## Freie Zeropage-Adressen

Das Programmieren in Assembler ist nicht möglich ohne Kenntnis wichtiger Adressen und Speicherbereiche eines

Computers. Der Prozessor des C16 bietet die sehr elegante indirekte Adressierung. Wie Sie wissen, ist es jedoch zur Verwendung dieser Adressierungsart notwendig, Pointer in der Zeropage des Computers abzulegen. Da – unabhängig von der indirekten Adressierung – die Zeropage die schnellste Form der Adressierung gestattet, die sogenannte »Zeropage-Adressierung«, wird man sie in zeitkritischen Routinen (Grafik!) ausgiebig verwenden, um möglichst viele Variablen darin abzulegen, auf die häufig zugegriffen werden muß. Der erste Schritt beim Kennenlernen des C16 besteht somit darin, herauszubekommen, welche Speicherplätze in der Zeropage unbenutzt sind und für eigene Zwecke zur Verfügung stehen.

Zuerst wäre der Bereich \$D8 bis \$E8 zu nennen. Dieser Bereich ist ungenutzt und steht sogar speziell für Anwendungssoftware zur Verfügung. In diesem Bereich können Sie alle Variablen ablegen, die ständig unverändert zur Verfügung stehen sollen. Alle weiteren Bereiche, die ich vorstelle, werden vom Basic-Interpreter beziehungsweise vom Betriebssystem benutzt und sind daher nicht unbedingt vorm Überschriften geschützt.

Ein Beispiel hierfür ist der Bereich \$61 bis \$70, die Fließkommaakkumulatoren eins und zwei. Dieser recht große zusammenhängende Bereich wird vom Basic-Interpreter für Berechnungen aller Art benutzt, übrigens keineswegs nur für Berechnungen mit reellen Zahlen, wie man aufgrund des Namens annehmen könnte, sondern ebenso für Berechnungen mit Integerzahlen. Wenn Sie keine reinen Assemblerprogramme schreiben, sondern Assembler Routinen als Ersatz für besonders zeitkritische Basic-Programmteile verwenden, seien Sie bitte vorsichtig: Wenn die betreffende Routine vom Basic-Programm aufgerufen wird und Werte in die Fließkommaakkumulatoren schreibt, dürfen Sie getrost davon ausgehen, daß nach der Rückkehr ins Basic-Programm und vor dem nächsten Aufruf der Routine diese Werte durch den Basic-Interpreter überschrieben werden. Speichern Sie daher in diesem Bereich bitte keine Werte, die permanent – auch nach der Rückkehr ins Basic-Programm – erhalten bleiben sollen.

Die beschriebenen Bereiche stellen ausreichend Platz für Pointer und oft benötigte Variablen zur Verfügung. Sollten Sie in Ihren Programmen jedoch Tabellen verwalten müssen, oder wollen Sie vor Beginn des Basic-RAMs gar eine komplette kleinere Maschinenroutine ablegen, benötigen Sie größere zusammenhängende Speicherbereiche. Schauen wir uns den Speicher von \$FF (Ende Zeropage) aufwärts an.

## Größere freie Speicherbereiche

Vor Beginn des Basic-Bereichs bietet der C16 mehrere Speicherbereiche, die für Tabellen und eventuell sogar für kleine Maschinenroutinen genutzt werden können:

1. \$0200 bis \$02AC: Ein riesiger uns zur Verfügung stehender Bereich, der normalerweise vom Basic-Interpreter und vom DOS als Eingabepuffer verwendet wird (zum Beispiel INPUT #-Befehl).

2. \$025D bis \$02AC: Diesen Bereich verwendet ebenfalls das DOS. Er darf nur benutzt werden, wenn im Moment des Aufrufs der Maschinenroutine keine (!) Kanäle geöffnet sind. Die vom DOS benötigten Parameter (Filename, Sekundäradresse, Geräteadresse etc.) werden sonst überschrieben.

3. \$0332 bis \$03F6: Kassettenspeicher und Zähler für Kassettenspeicherung. Wenn Sie in diesem Puffer Daten ablegen, bleiben sie erhalten, bis die nächste Kassettenspeicherung erfolgt. Damit eignet sich dieser Bereich zum Beispiel für kleinere Maschinenroutinen (wenn das Basic-Programm keine Kassettenspeicherung durchführt!).

4. \$03F7 bis \$0436: RS232-Puffer. Wenn Sie die RS232-Schnittstelle nicht verwenden (zum Beispiel für den Betrieb eines Modems), wird dieser Bereich völlig unbenutzt bleiben und ist daher hervorragend für Daten geeignet, die

permanent erhalten bleiben müssen und nicht durch Basic-Interpreter oder Betriebssystem überschrieben werden dürfen.

5. \$055F bis \$05E6: In diesem Bereich sind die Zeichenketten abgelegt, mit denen die Funktionstasten belegt sind. Sie können ihn für eigene Zwecke benutzen, wenn Ihr Basic-Programm die Funktionstasten nicht benötigt und deren Belegung daher überschrieben werden darf.

Wie Sie sehen, bietet der C 16 genug freie Speicherbereiche für die Ablage von Tabellen, Pointern und kleinen Assembler-routinen, sowohl in der Zeropage als auch dahinter, obwohl ich mich auf die größten zusammenhängenden Bereiche beschränkt habe.

Wenn Sie kleine Maschinenprogramme in diesen Bereichen ablegen wollen, beschränken Sie sich bitte auf den Kassettenspeicher, den RS232-Puffer und den Funktionstastenspeicher. In letzterem sind Daten vollkommen sicher, in den beiden anderen Bereichen besteht keine Überschreibgefahr, solange weder Kassettenspeichervorgänge noch die RS232-Schnittstelle benutzt wird.

#### Farbspeicher, Bildschirmspeicher, Basic-RAM

Ebenso wie bei vielen meiner Kollegen bestand meine erste Assembler-»Tat« im Beschreiben des Bildschirms mit unsinnigen Zeichen, zum Beispiel Füllen mit lauter Buchstaben. Da ich damals noch eine Scheu vor der Verwendung von Betriebssystemroutinen besaß, wurde jedes Zeichen von mir einzeln in den Bildschirmspeicher »gePOKEt« und die Farbe in der entsprechenden Speicherzelle des Farb-RAMs gesetzt. Für den Fall, daß Sie ebenfalls den Bildschirmspeicher direkt ansprechen wollen, stelle ich Ihnen nun die wichtigsten benötigten Adressen vor:

Farbspeicher: \$0800 bis \$0BFF

Bildschirmspeicher: \$0C00 bis \$0FFF

Wichtig sind nur die ersten 1000 Byte des jeweiligen Bereichs (25 Spalten x 40 Zeichen = 1000 Zeichen). Daß beide Bereiche dennoch ein volles Kilobyte umfassen, ist mitunter sehr vorteilhaft, zum Beispiel wenn der gesamte Bildschirm gelöscht oder mit einem bestimmten Zeichen beschrieben werden soll. In diesen Fällen können Sie mit einem recht einfachen Programm volle vier Seiten, das heißt 1024 Byte, beschreiben. Da der relevante Teil des Bildschirms beziehungsweise Farbspeichers nur 1000 Byte umfaßt, beschreiben Sie zwar mehr Speicherzellen als unbedingt nötig, das Programm wird dadurch jedoch erheblich vereinfacht und kürzer.

Das folgende Programmbeispiel (Listing 1) füllt den Bildschirm des C 16 mit dem Zeichen »A«. Geben Sie es bitte mit dem eingebauten Monitor ein, und starten Sie es anschließend mit »SYS 1015«:

```
. 03f7 a9 00 lda #$00
. 03f9 85 d8 sta $d8
. 03fb 85 da sta $da
. 03fd a9 08 lda #$08
. 03ff 85 d9 sta $d9
. 0401 a9 0c lda #$0c
. 0403 85 db sta $db
. 0405 a9 01 lda #$01
. 0407 a2 04 ldx #$04
. 0409 a0 00 ldy #$00
. 040b 91 d8 sta ($d8),y
. 040d 91 da sta ($da),y
. 040f 88 dey
. 0410 d0 f9 bne $040b
. 0412 e6 d9 inc $d9
. 0414 e6 db inc $db
. 0416 ca dex
. 0417 d0 f2 bne $040b
. 0419 60 rts
```

Listing 1.  
»Fill Screen«

Programmablauf: Das Programm liegt im RS232-Puffer und ist damit vor dem Überschreiben geschützt, solange diese Schnittstelle nicht benutzt wird. Da das Programm mit indirekt indizierter Adressierung arbeitet, werden zuerst zwei Pointer angelegt: ein Pointer auf den Beginn des Farb-RAMs (\$0800) und ein zweiter Pointer auf den Beginn des Bildschirm-RAMs (\$0C00). Der erste Pointer kommt nach \$D8/\$D9, der zweite Pointer nach \$DA/\$DB, das heißt in den speziell für Anwendungssoftware reservierten freien Zeropage-Bereich.

Nachdem die Pointer angelegt wurden, werden sowohl Bildschirm- als auch Farbspeicher mit dem Wert eins beschrieben, und zwar jeweils vier Seiten (X-Register wird mit vier geladen). Das Resultat besteht darin, daß der Bildschirm mit dem Zeichen »A« gefüllt wird (1=Bildschirmcode von »A«).

Speichern Sie dieses Programm aus dem Monitor heraus bitte mit »S" FILL SCREEN",8,03F7,041A« auf Diskette beziehungsweise mit »S" FILL SCREEN",1,03F7,041A« auf Kassette ab, da wir es in Kürze wieder verwenden werden.

#### Einbindung von Maschinenroutinen

Das vorgestellte Programm paßt gerade noch in den RS232-Puffer. Bei größeren Programmen stellt sich jedoch die Frage, wo wir sie im Speicher ablegen können. Für Maschinenprogramme, die ein oder gar mehrere Kilobyte umfassen, steht nur ein zusammenhängender Speicherbereich zur Verfügung, der groß genug ist: der Basic-Speicher selbst.

Der Basic-Speicherbereich beginnt ab \$1000. Wo er endet, hängt von mehreren Faktoren ab:

1. Normalerweise endet der Basic-Bereich bei \$3FFF.
2. Wurde die hochauflösende Grafik eingeschaltet, wofür zusätzlich Speicherplatz benötigt wird, endet Basic bei \$17FF.
3. Wird eine Speichererweiterung verwendet, hängt das Ende des Basic-Speichers von der Größe dieser Erweiterung ab.

Unveränderlich ist somit nur der Beginn des Basic-Speichers. Wenn Sie Wert darauf legen, daß Ihre Maschinenprogramme auch mit Grafik und Speichererweiterungen zusammenarbeiten, benutzen Sie bitte die Zeropage-Adressen \$33 und \$34, um das Ende des Basic-Bereichs zu ermitteln. Diese Adresse stellt einen Pointer (\$33=Low/\$34=High) dar, der auf das aktuelle Ende des verfügbaren Speichers zeigt. Wenn keine Erweiterung verwendet wird und die hochauflösende Grafik nicht eingeschaltet wurde, zeigt dieser Pointer auf die Adresse \$3FFF (probieren Sie's aus). Verwenden Sie in Ihren Programmen möglichst immer diesen Pointer anstelle einer absoluten Adresse, die sich – abhängig von der Grafik und Speichererweiterungen – ändern kann.

Wenn Sie reine Assemblerprogramme schreiben, zum Beispiel eine Textverarbeitung oder ein Malprogramm, benutzen Sie einfach den Basic-Speicher ab \$1000. Häufiger dürfte jedoch die Verwendung von Assembler-routinen sein, die mit einem Basic-Programm zusammenarbeiten sollen, zum Beispiel Eingabe- oder Sortier-routinen.

In einem solchen Fall legt man üblicherweise die Assembler-routinen ans Ende des Basic-Speichers und setzt den Zeiger auf dieses Ende (\$33/\$34) herab, um ein Überschreiben durch Strings zu verhindern, die der Basic-Interpreter leider ebenfalls vom Ende des Basic-Speichers nach unten anlegt.

Im Falle des C 16 ist diese Vorgehensweise etwas problematisch, da das Einschalten der hochauflösenden Grafik genügt, um die Routinen durch den Grafikbildschirm zu überschreiben. Eine flexiblere Methode, die sich mit der Grafik und auch mit eventuellen Speichererweiterungen verträgt, besteht darin, die Routinen vor (!) das Basic-Programm zu legen. Die Adressen \$2B/\$2C enthalten einen Zeiger auf

den Basic-Anfang. Diesen Zeiger können wir verändern und uns vor dem Basic-Programm beliebig Platz schaffen. Die Vorgehensweise:

1. Zeiger auf Basic-Anfang erhöhen.
  2. Null in die neue Startadresse »POKE«, da am Basic-Anfang immer (!) eine Null stehen muß.
  3. »NEW« eingeben, damit der Basic-Interpreter alle weiteren Zeiger entsprechend korrigiert.
  4. Maschinenroutinen und Basic-Programm laden (sofern erstere nicht bereits in Form von DATA-Zeilen im Basic-Programm enthalten sind).
- Beispiel: 1 KByte vorm Basic-Anfang für Maschinenroutinen reservieren (Listing 2):

```
10 POKE 44,PEEK(44)+4
20 POKE(256*PEEK(44)),0
30 NEW
```

Listing 2. »INT«

In Zeile 10 wird der Basic-Anfang um 1 KByte heraufgesetzt, das heißt in das High-Byte des Pointers auf den Basic-Start (\$33/\$34 = dezimal 43/44) wird der bisherige Wert plus vier geschrieben. Die Erhöhung des High-Bytes um den Wert vier entspricht vier Seiten, also einem Kilobyte.

Anschließend wird in die Speicherstelle, auf die das High-Byte nun zeigt, eine Null »gePOKEt«, und durch »NEW« werden zuletzt alle übrigen Zeiger korrigiert. Wenn Sie dieses Programm eingeben, starten und anschließend mit »FRE(0)« den freien Speicherplatz ermitteln, werden Sie feststellen, daß ein Kilobyte weniger als gewohnt zur Verfügung steht, jenes Kilobyte, das wir soeben für Maschinenprogramme reserviert haben.

Speichern Sie das Programm bitte unter dem Namen »INIT« auf Diskette oder Kassette ab.

Die beschriebene Vorgehensweise zur Einbindung von großen Assemblerprogrammen in Basic-Programme wollen wir nun an einem Beispiel erproben: Laden Sie bitte »INIT« und starten Sie es (sofern Sie das nicht bereits im vorigen Schritt getan hatten). Gehen Sie in den Monitor und laden Sie »FILL SCREEN«. Verschieben Sie die Routine bitte nun mit:

T 03F7 041A 1000

Die Routine befindet sich nun in unserem für Maschinenprogramme reservierten Ein-Kilobyte-Bereich, der ab \$1000 beginnt. Geben Sie folgendes Programm ein (Listing 3):

```
10 FOR i=1 to 10
20 PRINT CHR$(147)
30 PRINT "zum aufruf der routine taste druecken"
40 GETKEY A$
50 SYS 4096
60 FOR II=1 TO 1000:NEXT
70 NEXT
```

Listing 3. »INT-Demo«

Wie Sie sehen, ist die Einbindung wunderbar gelungen: Ab \$1000 (= dezimal 4096) befindet sich die Maschinenroutine im Speicher, ab \$2000 beginnt das Basic-Programm, das die Routine mit »SYS 4096« auf Tastendruck aufruft, woraufhin diese den Bildschirm mit »A« beschreibt und anschließend in das Basic-Programm zurückkehrt.

Die Verwendung der Standardbetriebssystemroutinen hat den Nachteil, daß diese sehr allgemein und flexibel, dadurch jedoch für bestimmte Zwecke manchmal etwas zu langsam sind. In extrem zeitkritischen Routinen kommen Sie nicht umhin, sich Ihre eigenen Routinen – zum Beispiel zur Bildschirmausgabe – zu schreiben, die weniger flexibel, dafür jedoch schneller sind. Sollten Sie jemals in die Verlegenheit kommen, C16-Assemblerprogramme auf den C64 oder den

C128 umschreiben zu müssen, werden Sie jedoch dankbar für jede von Ihnen verwendete Betriebssystemroutine sein. Die Standardein-/ausgabe-Routinen sind auf allen drei Computern identisch. Identisch bedeutet nicht, daß die Programme selbst gleich sind, sondern die »Benutzeroberfläche«. Das heißt, daß Routinen die gleichen Auswirkungen haben, die gleichen Parameter übergeben werden müssen und sogar, daß die Einsprungadressen identisch sind.

Letzteres dank der sogenannten »Kernel-Sprungtabelle«, die jeder der genannten drei Computer besitzt. Die Betriebssystemroutinen befinden sich zwar von Computer zu Computer in verschiedenen Speicherbereichen, werden jedoch nie direkt, sondern immer indirekt über diese Sprungtabelle angesprochen. Ein Beispiel:

Die Routine BSOUT gibt ein Zeichen auf dem Bildschirm aus. Aufgerufen wird sie mit einem: JSR \$FFD2, und zwar auf jedem (!) Commodore-Computer. An Adresse \$FFD2 befindet sich ein Sprungbefehl, zum Beispiel JMP \$1000, JMP \$2000 oder JMP \$3000. An welcher Adresse sich die Routine selbst befindet, ist dank der Sprungtabelle für den Benutzer uninteressant, jedenfalls solange er sich an die »offizielle« Methode hält, Betriebssystemroutinen keinesfalls direkt, sondern immer über die Sprungtabelle aufzurufen.

## Die wichtigsten Betriebssystemroutinen

Im folgenden werde ich zu jedem Beispielpogramm die entsprechenden Basic-Programmzeilen vorstellen, um vor allem den Umsteigern, die von Basic her kommen, den Umgang mit den Betriebssystemroutinen zu erleichtern.

### GETIN (\$FFE4)

Die Routine GETIN (Listing 4) liest ein Zeichen von der Tastatur, wartet jedoch nicht darauf, daß eine Taste gedrückt wird, analog dem Basic-Befehl GET. Der ASCII-Code des Zeichens wird im Akku übergeben. Wurde kein Zeichen gelesen, enthält der Akku den Wert Null. Durch das Testen auf Null kann eine Schleife realisiert werden, die erst nach der Eingabe eines Zeichens verlassen wird.

```
Basic:
10 GET A$:IF A$="" THEN 10
```

```
oder auch:
10 GETKEY A$
```

```
Assembler:
. 03f7 20 e4 ff jsr $ffe4
. 03fa f0 fb beq $03f7
. 03fc 60 rts
```

Listing 4. »GETIN«

Starten Sie diese Routine mit »SYS 1015«. Sie kehren nach dem Betätigen einer beliebigen Taste ins Basic zurück.

Mit GETIN lassen sich außer der Eingabe eines einzigen Zeichens auch komplexere Eingaben durchführen. Alle professionellen Eingaberoutinen, die den Basic-Befehl INPUT aufgrund seiner Nachteile (Bildschirm kann gelöscht werden, während der Eingabe sind keinerlei Eingabeprüfungen möglich und so weiter) ersetzen sollen, verwenden diese Betriebssystemroutine.

### BSOUT (\$FFD2)

Gibt ein Zeichen, das im Akku übergeben wird, auf dem momentanen Ausgabegerät aus, normalerweise dem Bildschirm (Listing 5). Bei der Bildschirmausgabe ist zu beachten, daß das Zeichen an der momentanen Cursorposition ausgegeben und dieser danach automatisch auf das nächste Zeichen gesetzt wird.

Basic:

```
10 PRINT "A";
20 PRINT "B";
```

Assembler:

```
. 03f7 a9 41 lda #$41
. 03f9 20 d2 ff jsr $ffd2
. 03fc a9 42 lda #$42
. 03fe 20 d2 ff jsr $ffd2
. 0401 60 rts
```

Listing 5. »BSOUT«

Ebenso wie das Basic-Programm gibt auch die Assembler-Routine nach dem Aufruf mit »SYS 1015« die Zeichen »A« und »B« (ASCII-Code: \$41/\$42) auf dem Bildschirm aus.

Wenn Sie bereits länger in Assembler programmieren, wird für Sie ein kleiner Trick bereits selbstverständlich sein, den ich hier verraten möchte: Da alle Betriebssystemroutinen als Unterprogramme geschrieben sind und mit einem »RTS« als letzten Befehl enden, können Sie Ihre Programme kürzen, indem Sie einen »RTS« einsparen, der nach dem Aufruf einer Betriebssystemroutine folgt, und diese anstatt mit »JSR...« mit »JMP...« aufrufen (Listing 6).

```
. 03f7 a9 41 lda #$41
. 03f9 20 d2 ff jsr $ffd2
. 03fc a9 42 lda #$42
. 03fe 4c d2 ff jmp $ffd2
```

Listing 6. »BSOUT ohne RTS«

#### PLOT (\$FFF0)

Setzt den Cursor an eine übergebene Bildschirmposition oder übergibt die momentane Cursorposition. Ein Basic-Äquivalent zu dieser Routine existiert nicht. In beiden Fällen wird das X-Register zur Übergabe der Zeile und das Y-Register zur Übergabe der Spalte verwendet:

1. Cursor setzen:

- Carryflag löschen
- X und Y laden
- Routine aufrufen

Cursor auf Zeile 10, Spalte 5 setzen (Listing 7):

```
. 03f7 a2 0a ldx #$0a
. 03f9 a0 05 ldy #$05
. 03fb 18 clc
. 03fc 20 f0 ff jsr $fff0
. 03ff 60 rts
```

Listing 7. Cursorposition setzen

Rufen Sie die Routine mit »SYS 1015:PRINT "HALLO"« auf. Die Zeichenfolge »HALLO« wird ab Spalte 5 von Zeile 10 ausgegeben werden.

Cursorposition holen (Listing 8):

```
. 03f7 38 sec
. 03f8 20 f0 ff jsr $fff0
. 03fb 00 brk
```

Listing 8. Cursorposition holen

Zeile und Spalte werden in X und Y übergeben. Die entsprechenden Registerinhalte werden Ihnen nach dem BREAK angezeigt. Beachten Sie jedoch, daß der Basic-Interpreter einen Zeilenvorschub ausführt. Wegen dieses Zeilenvorschubs enthält das X-Register die Zeile+1, in der sich der Cursor beim Aufruf befand, und das Y-Register den Spaltenwert Null.

## Anwendung der bisher beschriebenen Routinen

Mit den beschriebenen Routinen läßt sich bereits eine Menge anfangen. Ein Beispiel: Sie schreiben ein Malprogramm in Assembler. Der Benutzer will ein Bild, das gerade gemalt wurde, komplett löschen. Da dieser Vorgang unwiderprüflich ist, stellen Sie eine sogenannte »Sicherheitsabfrage«. Sie fragen den Benutzer:

»SIND SIE SICHER (J/N) ?«

Die Bildschirmausgabe soll in Zeile 10 und Spalte 15 erscheinen, das heißt etwa in der Mitte des Bildschirms. Außer den Tasten »j« und »n« soll keine weitere zur Beantwortung der Frage zulässig sein, der GET-Befehl ist daher hervorragend zur Lösung des Problems geeignet (Listing 9).

Basic:

```
10 PRINT CHR$(19);:REM CURSOR HOME
20 FOR I=1 TO 10
30 PRINT CHR$(17);:REM CURSOR DOWN
40 NEXT I
50 PRINT TAB(15)"SIND SIE SICHER (J/N) ?"
60 GETKEY A$
70 IF A$="J" THEN GOTO 100:REM LOESCHEN
80 IF A$="N" THEN GOTO 200:REM NICHT LOESCHEN
90 GOTO 60:REM FALSCH EINGABE
```

Assembler:

```
. 03f7 a2 0a ldx #$0a
. 03f9 a0 0e ldy #$0e
. 03fb 18 clc
. 03fc 20 f0 ff jsr $fff0

. 03ff a2 00 ldx #$00
. 0401 bd 1c 04 lda $041c,x
. 0404 20 d2 ff jsr $ffd2
. 0407 e8 inx
. 0408 e0 23 cpx #$23
. 040a d0 f5 bne $0401

. 040c 20 e4 ff jsr $ffe4
. 040f c9 4a cmp #$4a
. 0411 f0 07 beq $041a
. 0413 c9 4e cmp #$4e
. 0415 f0 04 beq $041b
. 0417 4c 0c 04 jmp $040c
. 041a 60 rts
. 041b 60 rts
```

```
)041c 53 49 4e 44 20 53 49 45 20 53 49 43 48 45 52
20 28 4a 2f 4e 29 20 3f
```

Listing 9. Anwendungsbeispiel

Erläuterung:

Im ersten Programmabschnitt wird der Cursor auf Zeile 10 und Spalte 5 gesetzt. Anschließend wird die Zeichenkette »sind sie sicher (j/n) ?«, die 23 Zeichen enthält (X wird mit 23 verglichen), ab dieser Position auf dem Bildschirm ausgegeben. Die Zeichenkette selbst befindet sich im ASCII-Format am Ende des Programms. Geben Sie sie bitte mit Hilfe des Monitor-Befehls (>) ein, mit dem direkt Bytes verändert werden können.

Der letzte Programmteil besteht aus einer Routine, die ein Zeichen von der Tastatur liest, mit den ASCII-Codes der Zeichen »j« beziehungsweise »n« vergleicht (\$4A/\$4E) und zum

entsprechenden Programmteil verzweigt. Wurde keines der beiden Zeichen eingegeben, wird die Tastaturabfrage wiederholt.

Ein Tip: in Assemblerprogrammen ist es sehr oft notwendig, Zeichenketten auf dem Bildschirm auszugeben. Anstatt wie im vorgestellten Beispielprogramm die Anzahl der auszugebenden Zeichen zu zählen und das Ausgabeende anhand des Wertes von X zu erkennen, können Sie sich das Programmieren erleichtern, wenn Sie an jede Zeichenkette eine »Endemarke« anfügen, zum Beispiel das Byte \$00, und auf diese Endemarke prüfen (Listing 10). Das Abzählen der einzelnen Zeichen einer Zeichenkette wird dadurch überflüssig:

```

loop    ldx #$00
        lda char,x
        beq weiter
        jsr $ffdb
        inx
        jmp loop

weiter  ...
        ...
        ...
        rts

char    Zeichenkette im ASCII-Format+$00

```

**Listing 10. Stringausgabe mit Endemarke**

#### BASIN (\$FFCF)

Auch diese Routine (Listing 11) kann zur Eingabe von Zeichen verwendet werden. Bei Verwendung von BASIN erscheint der Cursor auf dem Bildschirm, und ein vom Benutzer eingegebenes Zeichen wird automatisch auf dem Bildschirm ausgegeben, analog dem Basic-Befehl INPUT. Ebenso wie beim INPUT-Befehl ist keine Möglichkeit gegeben, unsinnige Eingaben abzufangen, zum Beispiel das Löschen des Bildschirms. Eingegebene Zeichen werden beim Aufruf von BASIN im Akku übergeben.

Basic:

10 INPUT A\$

Assembler:

```

. 03f7 a2 00    ldx #$00
. 03f9 20 cf ff jsr $ffcf
. 03fc 9d 05 04 sta $0405,x
. 03ff e8       inx
. 0400 c9 0d    cmp #$0d
. 0402 d0 f5    bne $03f9
. 0404 60       rts

```

**Listing 11. »BASIN«**

Diese Routine liest solange Zeichen ein und speichert sie ab \$0405, bis der Benutzer »RETURN« drückt (ASCII-Code \$0D), arbeitet somit analog dem INPUT-Befehl.

Rufen Sie die Routine mit »SYS 1015« auf und geben Sie mehrere Zeichen ein, bevor Sie die Eingabe mit »RETURN« beenden. Sehen Sie sich anschließend mit dem Monitor den Bereich ab \$0405 an (m 0405), in dem die Eingabe abgelegt wurde.

Ich würde Ihnen aufgrund der beschriebenen Nachteile von BASIN empfehlen, sich mit Hilfe der GETIN-Routine eine eigene INPUT-Routine in Assembler zu schreiben und auf die Verwendung von BASIN zur Tastaturabfrage in professionellen Programmen möglichst zu verzichten.

## Programme laden/speichern

Das Betriebssystem des C 16 enthält eine sehr komfortable SAVE-Routine, mit der ein beliebiger Speicherbereich auf Kassette oder Diskette gespeichert werden kann. Die ebenfalls vorhandene LOAD-Routine ermöglicht das Laden eines Programms an eine beliebige Adresse. Diese Routinen können vielseitig verwendet werden, um zum Beispiel ein mit der hochauflösenden Grafik gemaltes Bild, eine Bildschirmmaske und andere Speicherbereiche zu speichern beziehungsweise zu laden. Vor der Benutzung dieser Routinen müssen jedoch die beiden Routinen SETNAM und SETLFS zur Vorbereitung aufgerufen werden.

#### SETLFS (\$FFBA)

Vor Benutzung der SAVE- oder LOAD-Routine müssen mit SETLFS eine logische Filenummer (Akku), die Geräteadresse (X-Register) und eine Sekundäradresse (Y-Register) angegeben werden (Listing 12).

Beispiel: logische Filenummer zwei, Gerät: Floppy, Sekundäradresse vier.

```

lda #$02    ;log. Filenummer
ldx #$08    ;Geräteadresse
ldy #$04    ;Sekundäradresse
jsr $ffba

```

**Listing 12. »SETLFS«**

#### SETNAM (\$FFBD)

Nachdem mit SETLFS die Fileparameter gesetzt wurden, muß mit der Routine SETNAM der Filename angegeben werden, indem im X- und im Y-Register die Adresse übergeben wird, an der dieser im Speicher abgelegt wurde, und im Akku die Länge des Namens (Listing 13):

1. Akku mit der Länge des Filenamens in Zeichen laden.
2. X-Register mit dem Low-Byte der Adresse laden.
3. Y-Register mit dem High-Byte der Adresse laden.
4. SETNAM aufrufen

Beispiel: Ein Speicherbereich soll unter dem Namen »test« gespeichert werden. Der Name »test« befindet sich (in ASCII-Form) an Adresse \$025E, der speziell für Filenamen reserviert ist.

```

lda #$04    ;Länge des Filenamens
ldx #$5e    ;Adresse: Low-Byte
ldy #$02    ;Adresse: High-Byte
jsr $ffbd

```

**Listing 13. »SETNAM«**

Bei der Speicherung auf Datasette ist es nicht notwendig, dem zu speichernden Programm einen Namen zu geben. Auch in diesem Sonderfall muß (!) jedoch vor Benutzung der LOAD- beziehungsweise SAVE-Routine SETNAM aufgerufen werden. Wenn kein Filename verwendet wird, muß im Akku der Wert Null übergeben werden (Länge des Namens: Null Zeichen). Die Inhalte von X- und Y-Register sind beliebig.

#### LOAD (\$FFD5)

Für das Laden eines Programms spielt die in der Vorbereitungsroutine SETLFS übergebene Sekundäradresse eine wichtige Rolle. Die ersten beiden Bytes jedes Programmfiles enthalten die Adresse, an der sich das Programm beim Speichern befand. Wurde vor dem Aufruf von LOAD die Sekundäradresse eins angegeben, wird das Programm an diese Ursprungsadresse geladen. Wurde eine Null als Sekundäradresse übergeben, so wird der Inhalt von X- und Y-Register beim Aufruf von LOAD als Adresse angesehen, an die das Programm geladen werden soll (X=Low-Byte, Y=High-Byte der Adresse).

Der Inhalt des Akkus entscheidet beim Aufruf von LOAD darüber, ob ein Programm geladen oder ein VERIFY durchgeführt werden soll (0=Laden, 1=Verifizieren).

Beispiele:

1. Programm »test« von Datasette an die Ursprungsadresse laden (Listing 14):

```
lda #$02 ;logische Filenummer
ldx #$01 ;Geräteadresse der Datasette
ldy #$01 ;absolut laden
jsr $ffba ;Fileparameter setzen

lda #$04 ;Länge des Filenamens
ldx #$5e ;Adresse low
ldy #$02 ;Adresse high
jsr $ffbd ;Filename setzen

lda #$00 ;Flag für LOAD
jsr $ffd5 ;LOAD-Routine aufrufen
```

**Listing 14. »LOAD« von Kassette**

2. Programm »test1« von Diskette an die Adresse \$1000 laden (Listing 15).

```
lda #$02 ;logische Filenummer
ldx #$08 ;Geräteadresse der Floppy
ldy #$00 ;an die angegebene Adresse laden
jsr $ffba ;Fileparameter setzen

lda #$05 ;Länge des Filenamens
ldx #$5e ;Adresse low
ldy #$02 ;Adresse high
jsr $ffbd ;Filename setzen

lda #$00 ;Flag für LOAD
ldx #$00 ;Ladeadresse low
ldy #$10 ;Ladeadresse high
jsr $ffd5 ;LOAD-Routine aufrufen
```

**Listing 15. »LOAD« von Diskette**

In beiden Beispielen wird davon ausgegangen, daß sich der Programmname »test« beziehungsweise »test1« in ASCII-Form ab Adresse \$025E im Speicher befindet. Ein weiteres Beispiel zum Laden eines Programms ohne Angabe eines Namens (nur bei der Datasette möglich):

3. Programm ohne Angabe eines Filenamens von Datasette absolut laden (Listing 16).

```
lda #$02 ;logische Filenummer
ldx #$04 ;Geräteadresse der Datasette
ldy #$01 ;absolut laden
jsr $ffba ;Fileparameter setzen

lda #$00 ;Länge des Filenamens
jsr $ffbd ;Filename setzen

lda #$00 ;Flag für LOAD
jsr $ffd5 ;LOAD-Routine aufrufen
```

**Listing 16. »LOAD« von Kassette ohne Angabe eines Filenamens**

### SAVE (\$FFD8)

Mit Hilfe der SAVE-Routine läßt sich ein beliebiger Speicherbereich auf Kassette oder Diskette speichern. Zuvor müssen wiederum SETLFS und SETNAM als Vorbereitungs-routinen aufgerufen werden. Bei SAVE müssen folgende Parameter übergeben werden:

1. Anfangsadresse des zu speichernden Bereichs. Hierzu muß in der Zeropage ein Zwei-Byte-Zeiger (Low/High) auf diese Adresse abgelegt werden. Bei Aufruf der Routine muß der Akku mit der Adresse geladen werden, an der sich der Zeiger befindet.

2. Endadresse +1 (!) des zu speichernden Bereichs. Diese Adresse wird mit dem X- und dem Y-Register übergeben (X=Adresse low, Y=Adresse high).

Beispiele:

1. \$1000 bis \$2000 unter dem Namen »test« auf Diskette speichern (Listing 17).

```
lda #$02 ;logische Filenummer
ldx #$08 ;Geräteadresse der Floppy
ldy #$02 ;beliebige Sekundäradresse
jsr $ffba ;Fileparameter setzen

lda #$04 ;Länge des Filenamens
ldx #$5e ;Adresse low
ldy #$02 ;Adresse high
jsr $ffbd ;Filename setzen

lda #$00 ;Anfangsadresse low
sta $d8
lda #$10 ;Anfangsadresse high
sta $d9
lda #$d8 ;Adresse des Zeigers
ldx #$01 ;Endadresse+1 low
ldy #$20 ;Endadresse+1 high
jsr $ffd8 ;SAVE-Routine
```

**Listing 17. »SAVE« auf Diskette**

2. \$1234 bis \$3456 unter dem Namen »test1« auf Datasette speichern (Listing 18).

```
lda #$02 ;logische Filenummer
ldx #$01 ;Geräteadresse der Datasette
ldy #$02 ;beliebige Sekundäradresse
jsr $ffba ;Fileparameter setzen

lda #$05 ;Länge des Filenamens
ldx #$5e ;Adresse low
ldy #$02 ;Adresse high
jsr $ffbd ;Filename setzen

lda #$34 ;Anfangsadresse low
sta $d8
lda #$12 ;Anfangsadresse high
sta $d9
lda #$d8 ;Adresse des Zeigers
ldx #$57 ;Endadresse+1 low
ldy #$34 ;Endadresse+1 high
jsr $ffd8 ;SAVE-Routine
```

**Listing 18. »SAVE« auf Kassette**

Anwendung: hochauflösende Grafik speichern und laden. Als praktische Anwendung der LOAD- und der SAVE-Routine will ich Ihnen nun zeigen, wie Sie den Grafikbildschirm speichern und laden können, der den Speicher von \$2000 bis \$3FFF belegt. Wir benötigen hierzu ein Basic- und ein Maschinenprogramm. Beide sind in einer Disketten- (Listing 19) und einer Kassettenversion (Listing 20) abgebildet.

Geben Sie bitte zuerst die Maschinenroutine mit Hilfe des Monitors ein. Sie speichert den Grafikbereich unter dem Namen »GRAPHIK« auf Datasette beziehungsweise Diskette und lädt den Grafikbildschirm wieder in diesen Bereich.

```
. 03f7 a9 02 lda #$02
. 03f9 a2 08 ldx #$08
. 03fb a0 02 ldy #$02
. 03fd 20 ba ff jsr $ffba
. 0400 a9 07 lda #$07
. 0402 a2 5e ldx #$5e
. 0404 a0 02 ldy #$02
. 0406 20 bd ff jsr $ffbd
. 0409 a9 00 lda #$00
. 040b 85 d8 sta $d8
. 040d a9 20 lda #$20
. 040f 85 d9 sta $d9
. 0411 a9 d8 lda #$d8
. 0413 a2 00 ldx #$00
. 0415 a0 40 ldy #$40
. 0417 20 d8 ff jsr $ffd8
. 041a 60 rts
```

```
. 041b a9 02 lda #$02
. 041d a2 08 ldx #$08
. 041f a0 01 ldy #$01
. 0421 20 ba ff jsr $ffba
. 0424 a9 07 lda #$07
. 0426 a2 5e ldx #$5e
. 0428 a0 02 ldy #$02
. 042a 20 bd ff jsr $ffbd
. 042d a9 00 lda #$00
. 042f 20 d5 ff jsr $ffd5
. 0432 60 rts
```

Listing 19. »SAVE LOAD« auf und von Diskette

```
. 03f7 a9 02 lda #$02
. 03f9 a2 01 ldx #$01
. 03fb a0 02 ldy #$02
. 03fd 20 ba ff jsr $ffba
. 0400 a9 07 lda #$07
. 0402 a2 5e ldx #$5e
. 0404 a0 02 ldy #$02
. 0406 20 bd ff jsr $ffbd
. 0409 a9 00 lda #$00
. 040b 85 d8 sta $d8
. 040d a9 20 lda #$20
. 040f 85 d9 sta $d9
. 0411 a9 d8 lda #$d8
. 0413 a2 00 ldx #$00
. 0415 a0 40 ldy #$40
. 0417 20 d8 ff jsr $ffd8
. 041a 60 rts
```

```
. 041b a9 02 lda #$02
. 041d a2 01 ldx #$01
. 041f a0 01 ldy #$01
. 0421 20 ba ff jsr $ffba
. 0424 a9 07 lda #$07
. 0426 a2 5e ldx #$5e
. 0428 a0 02 ldy #$02
. 042a 20 bd ff jsr $ffbd
. 042d a9 00 lda #$00
. 042f 20 d5 ff jsr $ffd5
. 0432 60 rts
. 0433 00 brk
```

Listing 20. »SAVE LOAD« auf und von Kassette

Der einzige Unterschied zwischen der Disketten- und der Kassettenversion besteht in der unterschiedlichen Geräteangabe bei Aufruf der Routine SETFLS.

Geben Sie nun bitte die Disketten- beziehungsweise Kassettenversion des benötigten Basic-Programms ein (Listing 21, 22).

```
10 A$=GRAPHIK:REM FILENAME IN ASCII-
20 FOR I=1 TO LEN(A$):REM FORM NACH 606
30 POKE 605+I,ASC(MID$(A$,I,1))
40 NEXT
50 :
60 GRAPHIC1,1
70 BOX 1,10,10,200,100
80 BOX 1,50,50,300,190
90 PAINT 1,60,60
100 :
110 SYS 1015:REM GRAFIK SPEICHERN
120 SCNCILR:REM SCREEN LOESCHEN
130 SYS 1015+36:REM GRAFIK LADEN
140 GETKEY Aa$
150 GRAPHICO
```

Listing 21. Grafik aufrufen und Maschinenroutine aufrufen (Diskette)

```
10 A$=GRAPHIK:REM FILENAME IN ASCII-
20 FOR I=1 TO LEN(A$):REM FORM NACH 606
30 POKE 605+I,ASC(MID$(A$,I,1))
40 NEXT
50 :
60 GRAPHIC1,1
70 BOX 1,10,10,200,100
80 BOX 1,50,50,300,190
90 PAINT 1,60,60
100 :
110 CHAR 1,0,24,ZURUECKSPULEN/RECORD+PLAY DRUECKEN
120 SYS 1015:REM GRAFIK SPEICHERN
130 CHAR 1,0,24,ZURUECKSPULEN/TASTE DRUECKEN
140 GETKEY A$
150 SCNCILR:REM SCREEN LOESCHEN
160 CHAR 1,0,24,PLAY DRUECKEN
170 SYS 1015+36:REM GRAFIK LADEN
180 GETKEY A$
190 GRAPHICO
```

Listing 22. Grafik erzeugen und Maschinenroutine aufrufen (Kassette)

Das Basic-Programm besteht aus folgenden Teilen:

1. Filename in ASCII-Form ab 606 (= \$025E) ablegen
2. Grafik einschalten
3. Rechtecke malen und ausfüllen
4. SAVE-Routine aufrufen
5. Grafik löschen
6. Gespeicherte Grafik laden

Zusätzlich enthält die Kassettenversion noch Hinweise zur Bedienung der Datasette (Zurückspulen vor dem Speichern, Zurückspulen vor dem Laden, Play beziehungsweise Record+Play drücken).

Sollten Sie eine Floppy besitzen, können Sie beobachten, wie sich die Grafik während des Ladevorgangs langsam aufbaut. Bei Verwendung einer Datasette wird bekanntlich der Bildschirm während Kassettenoperationen »weggeklappt«, so daß Sie die Grafik erst nach dem vollständigen Ladevorgang auf dem Bildschirm sehen. Kassettenbesitzer sollten sich in Geduld üben, da das Speichern beziehungsweise Laden einen jeweils acht Kilobyte großen Speicherbereich umfaßt und daher etwa fünf Minuten dauern wird.

(S. Baloui/ah)

## Tabellarische Übersicht

## Freie Speicherbereiche:

Zeropage: \$D8 bis \$E8 (Anwendersoftware)  
 \$61 bis \$70 (Fließkommaakkumulatoren)

## Sonstige Bereiche:

\$0200 bis \$02AC (Basic-Eingabepuffer/DOS-Arbeitsbereich)  
 \$025D bis \$02AC (aktuelle DOS-Parameter, zum Beispiel Filenamen)  
 \$0332 bis \$03F6 (Kassettenpuffer)  
 \$03F7 bis \$0436 (RS232-Puffer)  
 \$055F bis \$05E6 (Funktionstastenbelegung)

## Speicherbereiche/Zeiger:

Farbspeicher :\$0800 bis \$0BFF  
 Bildschirmspeicher :\$0C00 bis \$0FFF  
 Zeiger Basic-Anfang (Standard \$1000) :\$2B/\$2C  
 Zeiger Basic-Ende (Standard \$3FFF) :\$33/\$34

## Betriebssystemroutinen:

| Adresse | Label  | Funktion                                                                                                                       | Übergabeparameter                                                               |
|---------|--------|--------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| \$FFE4  | GETIN  | Zeichen holen                                                                                                                  | keine; Zeichenübergabe im Akku                                                  |
| \$FFCF  | BASIN  | Zeichen holen<br>bis RETURN                                                                                                    | keine; Zeichenübergabe im Akku                                                  |
| \$FFD2  | BSOUT  | Zeichen ausgeben                                                                                                               | Zeichen im Akku übergeben                                                       |
| \$FFF0  | PLOT   | Cursor setzen/<br>Position holen                                                                                               | Zeile(X)/Spalte(Y)/Carry Clear<br>Carry Set/Positionsübergabe in X/Y            |
| \$FFBA  | SETLFS | Filepar. setzen                                                                                                                | Filenr.(A)/Gerät(X)/Sek.Adresse(Y)                                              |
| \$FFBD  | SETNAM | Filename setzen                                                                                                                | Länge Name(A)/Adresse Name(X/Y)                                                 |
| \$FFD5  | LOAD   | Programm laden/<br>Ladeadresse Verify durchführen<br>(X/Y) bei relativem Laden, absolutes Laden<br>bei Sek.-Adresse 1 (SETLFS) | Load(0)-Verify(1)- Flag(A)/<br>Ladeadresse                                      |
| \$FFD8  | SAVE   | Programm speichern                                                                                                             | Zeropage-Adresse des Zeigers auf die<br>Anfangsadresse(A)/<br>Endadresse+1(X/Y) |

## Reaktionstest

Mit diesem Einzeiler können Sie Ihre Reaktionszeit messen. Um die Länge einer Basic-Zeile nicht zu überschreiten, müssen einige Befehle in abgekürzter Schreibweise eingegeben werden. Nach dem Start des Einzeilers erscheint nach einiger Zeit das Signal »T=«. Nun müssen Sie so schnell wie möglich eine beliebige Taste drücken. Nachdem die Reaktionszeit (in Sekunden) auf dem Bildschirm erschienen ist, beginnt auch gleich der nächste Test. Etwaigem Schummeln wurde vorgebeugt: Beim frühzeitigen Drücken einer Taste wird der Test abgebrochen.

```
2 FOR=0TORND (1) *2500+900:NEXT:GETA$:IFA$=
""THENPRINT "T=";:T=TI:GETKEYA$:PRINT (TI-T) /60:RUN
(Hannes Kaltenbach/tr)
```

## RESET ohne Reset-Schalter

Nach langem Herumprobieren wurde im Betriebssystem die Routine gefunden, die einen Reset-Schalter überflüssig macht. Die Routine wird vom Basic aus mit dem SYS-Befehl wie folgt angesprungen:

```
SYS 65529
```

Bei diesem Befehl werden alle Zeiger für das Basic und für das Betriebssystem in den Ausgangszustand zurückgesetzt (Basic-Kaltstart). Alle Maschinenprogramme bleiben jedoch erhalten und können wie gewohnt aufgerufen werden.

(J. Kunz/ah)

# Grafik und Sound mit dem C 16

**Der beste Computer nützt einem wenig, wenn man seine Fähigkeiten nicht einzusetzen weiß. Wir zeigen, wie man mit dem C16 Grafik und Sound programmiert.**

**A**ls Commodore 1984 die beiden Computer C 16/C 116 und Plus 4 auf den Markt brachte, ahnte wohl noch niemand, daß sich der C 16 gegen seinen größeren Bruder durchsetzen würde. Doch es geschah noch mehr: Der Plus 4 verschwand schon nach kurzer Zeit wieder von den Ladentischen. Er war für seine mäßige Leistung einfach zu teuer. Der C 16 hingegen konnte sich bis heute auf dem Markt halten. Dies liegt wohl an seinem geringen Preis: er ist im Moment schon für weit unter 300 Mark erhältlich. Für die Ausführung mit Gummitastatur des C 116 sind sogar zirka 50 Mark weniger zu berappen. Ob sich diese Einsparung jedoch lohnt, ist Ansichtssache. Wer zahlt nicht gerne 50 Mark mehr, um auf einer sehr guten und altbewährten Schreibmaschinentastatur ermüdungsfrei arbeiten zu können?

Wenn im folgenden Artikel vom C 16 die Rede ist, so ist damit natürlich ebenso der C 116 gemeint.

Der C 16 eignet sich besonders für Einsteiger, die mal in die Computerei hineinschnüffeln wollen. Der geringe Preis des Gerätes setzt die Hemmschwelle für Anfänger weit herunter.

Woran liegt es nun, daß der C 16 trotz seines geringen Preises nicht ein Renner wie beispielsweise der C 64 wurde?

Das einzige große Manko des C 16 ist sein geringer Speicherplatz. Er stellt dem Benutzer nur ganze 16 KByte zur Verfügung. Schaltet man in den Grafikmodus, dann bleiben sogar nur noch 2 KByte für das Programm. Abhilfen in Form von Speichererweiterungen sind vorhanden, aber nicht unbedingt billig. Von einigen Herstellern (außer Commodore) werden sie inzwischen angeboten.

Ansonsten läßt der C 16 kaum Wünsche offen (zumindest in seiner Preisklasse): Erstmals rüstet Commodore die Computer C 16 und Plus 4 mit dem verbesserten Basic 3.5 aus. Endlose Programme mit PEEK und POKE-Sequenzen gehören der Vergangenheit an. Sound und Grafik können bequem über Basic-Befehle programmiert werden (in diesem Punkt ist er sogar dem C 64 überlegen). In diesem Artikel wollen wir uns vor allem der Grafik- und Sound-Programmierung widmen.

Zur Grafik gehören natürlich auch die Farben. Der C 16 bietet im Gegensatz zum VC 20 oder C 64 erstmals die Möglichkeit, 121 verschiedene Farbstufen darzustellen. Diese setzen sich aus 16 Farben zusammen, von denen 15 in 8 Helligkeitsstufen dargestellt werden können. Für die sechzehnte Farbe (Schwarz) gibt es keine Helligkeitsunterschiede.