Action! noch schneller

Welcher Atari-Fan kennt ihn nicht, den superflinken ACTION!-Compiler von Optimized Systems Software (OSS)? Ohne Übertreibung ist das die beste und schnellste Programmiersprache für alle 8-Bit Computer von Atari, wenn nicht die schnellste für alle 6502-Computer überhaupt. Und die darf natürlich im Rahmen der Assemblerecke nicht vergessen werden.

Den Lesern der Assemblerecke ist ACTION! schon längst kein Unbekannter mehr, denn in der CK 10/85 haben wir bereits ein Musikprogramm vorgestellt, das vollkommen in ACTION! geschrieben war. Diesmal wird noch etwas tiefer gebohrt und gezeigt, wie man ACTION!-Programme noch kürzer und noch schneller machen kann. Außerdem gibt's als besonderes Bonbon eine Runtime-Package, mit der ACTION!-Programme auch ohne Steckmodul laufen.

Contra Basic

In vielen ACTION!-Listings, die in amerikanischen Zeitschriften zu finden waren, wurden eifrig PEEK- und POKE-Befehle eingesetzt. Das zeigt, dass die Autoren noch nicht erkannt haben, welche eleganten und leistungsfähigen Konstruktionen ACTION! anbietet und damit PEEK und POKE vollkommen überflüssig werden lässt. Nehmen wir als Beispiel nur die Abfrage eines Joysticks über die Speicherzelle 632 (STICK0). Ein Test auf die Mittelstellung könnte lauten:

```
IF PEEK(632) = 15 THEN …
```

Diese Konstruktion würde in Basic und auch in ACTION! funktionieren. Aber es geht viel eleganter. Legen Sie einfach eine BYTE-Variable auf die gewünschte Speicherzelle und benutzen Sie diese anstatt PEEK:

```
BYTE STICK0 = 632
IF STICK0 = 15 THEN …
```

Diese Variable ist nicht nur um einiges kürzer, sondern im Ablauf mindesten 10 mal schneller! Um das zu verstehen, muss man sich ansehen, wie der ACTION!-Compiler intern arbeitet. PEEK() ist innerhalb von ACTION! als Funktion vordefiniert und wird daher als Aufruf eines Unterprogramms per JSR übersetzt. Zuvor muss das Argument von PEEK noch in die Register des 6502-Prozessors geladen werden, so dass sich etwa folgendes Maschinenprogramm für den Zugriff auf die Speicherzelle 632 ergibt:

```
LDA #<632
LDX #>632
JSR PEEK
```

Das Unterprogramm PEEK liegt in der ACTION!-Cartridge. Ein Grund, warum solche Programme nur mit eingestecktem Modul funktionieren. Die zweite Möglichkeit mit der Variablen erzeugt nur folgenden Code:

```
LDA 632
```

Kürzer könnte man es in Assembler auch nicht programmieren. ACTION! braucht hier nur drei Bytes, um den Joystick zu lesen. Da kein Sprung in ein Unterprogramm anfällt und daher auch kein Unterprogramm bearbeitet werden muss, ist diese Möglichkeit die entschieden schnellere. Außerdem wird das Steckmodul nicht angesprochen. Darauf wird später noch mal genauer eingegangen.

Arrays

Auf der anderen Seite ist klar, dass mit diesen adressierten Variablen nicht jeder PEEK- oder POKE-Befehl ersetzbar ist. Einfachstes Beispiel wäre das Ändern eines der fünf Farbregister in Abhängigkeit von einer Variablen. Mit POKE(708+I,0) würde das mit I ausgewählte Farbregister mit der Farbe schwarz vorbelegt. Hier bieten sich die ungeheuer universalen Arrays des ACTION!-Compilers an. Wie bei einfachen Variablen können auch Felder auf feste Adressen gelegt werden und somit ist es möglich, eine Gruppe von Betriebssystem- oder Hardware-Registern als Array aufzufassen. Mit

```
BYTE ARRAY FARBE=708
```

definiert man die Schattenregister der Farbspeicher als ein Feld FARBE, und die obige POKE-Anweisung verkürzt sich zu

```
FARBE(I)=0
```

Aber damit haben wir den Compiler noch lange nicht ausgeschöpft. Mit einem kleinen Trick wird der erzeugte Code viel kürzer und schneller:

```
BYTE ARRAY FARBE(0)=708
```

Diese Definition sagt dem Compiler, dass er es mit einem kurzen Feld zu tun hat, das schnell adressiert werden kann. Im Objektprogramm kann der Compiler dann die X-Indizierung ausnutzen und die Anweisung

```
FARBE(INDEX)=0
```

so übersetzen:

```
LDA #0
LDX INDEX
STA FARBE,X
```

Wieder ist das Ergebnis selbst in Assembler kaum schneller zu programmieren. Lässt man jedoch den Zusatz "(0)" weg, wird der Compiler annehmen, dass es sich auch um ein längeres Feld handeln könnte und muss daher zur indirekten Adressierung greifen. Dazu wird bei der Definition ein Zeiger auf das Array angelegt, der bei der Verwendung erst in die Zero-Page verlagert werden muss. Nur dort kann der 6502 indirekte Adressierungsarten ausführen. Das Ergebnis sähe dann so aus:

```
ZEIGER .WORD FARBE
…
CLC
LDA ZEIGER
ADC INDEX
STA $AE
LDA ZEIGER+1
ADC #0
STA $AF
LDY #0
LDA #0
STA ($AE),Y
```

Sie sehen schon, das ist ungleich aufwendiger und daher auch viel langsamer (aber dennoch schneller als der ursprüngliche POKE-Befehl). Kurze Byte Arrays sollten Sie daher immer mit "(0)" definieren. Die lange "Zeiger"-Definition braucht man nur bei umfangreichen Feldern (ab 256 Bytes) oder falls das Feld während des Programmlaufes auf eine andere Adresse gelegt werden soll (auch das kann ACTION!).

Vielleicht noch eine Anmerkung für all diejenigen Leser, denen Felder ohne Längenangabe noch "spanisch" vorkommen. Der ACTION!-Befehl ARRAY wirkt wie die DIM-Anweisung in BASIC, nur DIM ist wesentlich weniger flexibel, da es dem Programmierer alle Arbeit abnimmt. Bei ARRAY kann man selbst bestimmen, wo das Feld im Speicher liegen soll, aber man ist auch für alle Konsequenzen selbst verantwortlich. So finden keinerlei Überprüfungen der Index-Grenzen statt, und wenn man sein Array so legt, dass lebenswichtige Zellen des Computers überschrieben werden, so hilft eben nur noch der Reset-Knopf. Damit erklären sich auch so anscheinend sinnlose Definitionen, wie es ein Feld mit der Länge Null oder ohne Längenangabe ist. Dies ist nur als ein Zeichen für den Compiler zu verstehen, der Programmierer muss schließlich selbst über die tatsächliche Länge wachen.

Pointer

Letzte und universelle Möglichkeit zum Ersetzen von PEEK und POKE bietet der Pointer. Derartige Zeiger-Variablen enthalten eine Adresse einer anderen Variablen oder einer beliebigen Speicherzelle. Basic-Kenner werden jetzt einwerfen, dass daran nichts Besonderes ist, schließlich kann jede normale Variable auch eine Adresse enthalten. Wozu also Pointer? Diese Spezial-Variablen haben Fähigkeiten, die über das normale Maß hinausgehen. So kann durch das Anfügen des Pointer-Symboles "^" der Inhalt der Variablen erreicht werden, auf die der Pointer zeigt. Sehen Sie sich das gleich an einem Beispiel an:

```
BYTE POINTER ZEIGER=708
BYTE F
F = ZEIGER^
ZEIGER ==+1
ZEIGER^ ==+2
```

Die erst Zeile definiert einen Pointer des Namens ZEIGER, der auf das erste Farbregister gerichtet ist. Der Variablen F wird der Inhalt von Speicherzelle 708 zugeordnet, das entspricht dem Befehl F=PEEK(708). Nun kommt der Trick: Der Pointer wird um eins erhöht, so dass er nun auf die Zelle 709 deutet. Die dritte Zeile erhöht schließlich den Inhalt von 709 um zwei, in Basic müsste man hier mit POKE 709,PEEK(709)+2 viel umständlicher vorgehen.

Anmerkungen: 1. Code für das Zeigerbeispiel korrigiert. Das Pointer-Symbol "^" muss in Zeile 4, nicht in Zeile 3. 2. Im Gegensatz zur normalen Initialisierung einer Variable mit eckigen Klammern (z.B. BYTE A=[5]; A enthält dann "5") werden Zeiger nur mit Gleichheitszeichen initialisiert (z.B. BYTE POINTER P=710 ; P ist dann gleich 710, zeigt also auf Speicherzelle 710)

Optimierung

Erneut haben wir den ACTION!-Compiler nur mit Halbgas gefahren. Jedem Assemblerprogrammierer wird bei der Besprechung der Pointer aufgefallen sein, dass es sich hierbei um eine reinrassige indirekte Adressierung handelt, und so etwas kann der 6502 nun mal eben nur in der Zero-Page. Wenn daher ein Pointer benutzt wird, muss der Compiler dafür sorgen, dass die Adresse in ein Arbeitsregister der Zero-Page geschrieben wird, und dann erst kann der Zugriff erfolgen. Warum also nicht gleich alle Pointer in die Zero-Page legen? Und es funktioniert! Der ACTION!-Compiler erkennt, wenn ein Pointer gleich dort ist, wo er sein sollte und arbeitet dann auch wesentlich schneller. Nimmt man an, dass ZEIGER in der Zero-Page liegt so wird ZEIGER=255 folgendermaßen übersetzt:

```
LDA #255
LDY #0
STA (ZEIGER),Y
```

Wüssten Sie, wie man es noch schneller und kürzer machen könnte? Bleibt nur noch das Problem, wie man Pointer in die Seite null bekommt, und wo Platz dafür ist. Schreibt man einfach BYTE POINTER PTR=$D4, so erhält der PTR zwar den Wert $D4, aber der Pointer selbst liegt an einem Ort, den ihm ACTION! zugedacht hat. Hilfe bringen hier die Definitionen ZEROPAG und RESTORE in Listing 1.

```
;**************************************
; ACTION! : Benutzung der Zero-Page
;
;PETER FINZEL                    1986
;**************************************

;Zero-Page Anweisungen
;=====================

DEFINE
ZEROPAG ="SET $0F=0 SET $0E=$D4",
RESTORE ="SET $0E=$491^ SET $493=0 SET $494=0"

;Demo-Programm
;=============

ZEROPAG                 ;Zeiger in Zero-
BYTE POINTER Bild_Ptr   ;page legen
CARD ENDE

RESTORE                 ;normales RAM

;Invertiert GR.8 Bildschirm
;==========================

PROC Invertiere( )
CARD Savmsc=$58         ;Video-RAM Adresse
BYTE i

Graphics(8+16)
FOR i=1 TO 20           ;20 mal invertieren
   DO                   ; Endlos- Schleife
   Bild_Ptr=Savmsc      ;Anfang und
   Ende = Savmsc+7680   ;Ende festlegen

   WHILE Bild_Ptr<Ende
        DO
        Bild_Ptr^==!$FF  ;Byte invertieren
        Bild_Ptr ==+1    ;Zeiger weiter
        OD
   OD
RETURN
```

Wer das ACTION!-Handbuch (Seite 144) genau gelesen hat, weiß, dass die Adresslage des erzeugten Objektprogramms mit den Speicherzellen $0E, $0F und $491, $492 gesteuert werden kann. Durch geschickte DEFINEs kann man sich quasi zusätzliche Befehle zum Verändern der Adressen schaffen. Die neue Anweisung ZEROPAG verlagert alle nachfolgenden Definitionen in die Zero-Page. Da es dort recht eng zugeht, sollten Sie nur wichtige und häufig benutzte Variablen dahin legen. Sie dürfen auch niemals vergessen, die Adresslage nach den Definitionen mit RESTORE wieder in einen "normalen" Bereich zurückzulegen. In Listing 1 finden Sie auch gleich ein Beispiel für die Definition eines Pointers in der Seite Null.

Wenn das Compilat später mit einem W-Befehl des Monitors gespeichert werden soll, wird erst ab RESTORE aufgezeichnet. Verwenden Sie daher diese Anweisungen bitte nur am Anfang eines Programms. Es hat auch keinen Sinn, Variablen in der Zero-Page vorzubesetzen. Probleme mit ZP-Variablen kann es geben, wenn im Programm gleichzeitig Ein-/Ausgabe von Zahlen, z.B. mit PrintC(), vorkommt. Hier hilft leider nur, wenn man seine eigenen Routinen zur Zahlenausgabe verwendet (s. auch CK 11/85).

Nebenbei bemerkt ergibt sich auch für normale (BYTE, CARD) Variablen ein Vorteil in Ausführungszeit und Speicherplatz, wenn diese in die Zero-Page gelegt werden. Der Compiler bemerkt dies und wendet auf solche Variablen die optimaleren Zero-Page Adressierungen an. Zum Beispiel konnte der im ACTION!-Handbuch abgedruckte Benchmark-Test noch beschleunigt werden. Mit abgeschaltetem Bildschirm dauert die Berechnung der ersten 1899 Primzahlen noch ca. 1 sec. Das ist schneller als es ein 520 ST unter Forth schafft! Man merkt eben, dass ACTION! von einem ganz ausgekochten 6502-Profi entwickelt wurde.

Ohne Modul?

Eines der schlimmsten Nachteile von ACTION! ist, dass das Steckmodul zum Ablauf eines Programms erforderlich ist, auch wenn es sich um bereits compilierte Programme handelt. Damit wird es unmöglich, die eigenen Programmierkunstwerke an Freunde weiterzugeben, die keine der immer noch ca. 250,-- DM teuren ACTION!-Module haben. Ganz zu schweigen von der Entwicklung kommerzieller Programme, wozu sich ACTION! aber vorzüglich eignen würde. Der Grund für diesen Sachverhalt ist darin zu suchen, dass ACTION! nicht alle im Programm verwendeten Funktionen auch in das Programm mit aufnimmt. Vielmehr sind im erzeugten Objektcode Unterprogrammsprünge in das ROM-Modul zu finden, und damit läuft das Programm eben nur mit dem Modul. Diese Sprünge ins ROM werden sowohl bei aufwendigeren Rechenbefehlen (*, /) als auch beim Aufruf von Funktionen oder eingebauten Prozeduren (wie STICK() oder SetBlock) verwendet.

Alle bisher vorgestellten Verfahren haben zusätzlich den Vorteil, dass sie das ACTION!-Modul nicht benötigen. Hätten Sie PEEK oder POKE oder gar STICK() oder SETCOLOR() verwendet, müsste die Cartridge wohl oder übel vorhanden sein. Oder man hat die sogenannte Runtime-Package, mit deren Hilfe Programme auch ohne Modul laufen. Man kann jedoch ein Programm in ACTION! so schreiben, dass es tatsächlich ohne Runtime-Package auskommt. Folgende Punkte sind dabei zu beachten:

1. Es darf keine der eingebauten Funktionen und Prozeduren verwendet werden. Das heißt kein PEEK, kein POKE und auch kein PRINT!
2. Multiplikationen sowie Divisionen dürfen nicht verwendet werden.
3. Die Schiebe-Operationen RSH und LSH dürfen nur noch auf Byte-Werte angewendet werden.
4. Beim Aufruf von Prozeduren und Funktionen dürfen nicht mehr als drei Bytes (oder ein Card und ein Byte) übergeben werden.

Wenn Sie diese Grundregeln peinlich genau beachten, läuft Ihr Programm auch ohne Modul. Natürlich ergeben sich damit einschneidende Abstriche. Ohne PRINT ist es nun mal schwierig, Texte und Zahlen auszugeben und ohne volle Parameterübergabe muss einiges umständlicher als sonst programmiert werden.

Runtime

Besser geht es natürlich mit eine Runtime-Package (RTP).Von OSS wurde zwar eine RTP angekündigt, die aber bei uns nie auf dem Markt erschienen. Gerüchten nach zu urteilen, hat es Probleme zwischen OSS uns Action Computer Services (dem Hersteller von ACTION!) gegeben. Damit bleibt dem geplagten Benutzer nur eine Lösung: Eine eigene Runtime-Package muss her. Und hier ist sie. Sie brauchen nur Listing 2 und 3 abzutippen und unter den angegebenen Filenamen abzuspeichern. Später schreiben Sie in Ihr Programm

```
INCLUDE "RUN1.ACT"
INCLUDE "RUN2.ACT"
```

und haben dann die RTP in Ihr Programm eingebunden.

Interessanterweise ist die Runtime-Package (RTP) auch in ACTION! geschrieben, wenn auch an den wesentlichen Stellen sog. Code-Blocks verwendet wurden. An dieser Stelle ist gleich zu vermerken, dass es sich um keine vollständige RTP handelt, sondern nur die wichtigsten Befehle implementiert wurden. Das ist schon ein Hinweis, wie flexibel die RTP in ACTION! verwendet werden kann. Nur diejenigen Features, die auch tatsächlich gebraucht werden, müssen in der RTP enthalten sein. Wenn sie ein Programm haben, das keinerlei I/O benötigt, dann reicht es, wenn sie RUN1.ACT hinzuziehen. Man kann sogar noch weiter gehen und Multiplikation und Divisions-Routinen aus RUN1 streichen, falls sie nicht gebraucht werden. In jedem Fall sollte aber die Parameter-Routine enthalten sein.

In RUN1.ACT sind die Grundrechenarten sowie die Shift-Befehle für Cards enthalten. Außerdem dürfen dank PAR() wieder mehr als drei Bytes an Prozeduren übergeben werden. RUN2.ACT stockt den Befehlsvorrat mit den wichtigsten I/O-Befehlen auf. Hier wurden OPEN und CLOSE implementiert, weiterhin sind PUT und GET sowie die PRINT-Routinen für Strings enthalten. Die Definitionen entsprechen den Angaben im Handbuch, nur bei der Fehlerbehandlung wurde ein anderer Weg eingeschlagen: Nach jeder I/O-Anweisung können Sie anhand der Byte-Variablen IOERR prüfen, ob sich ein Fehler eingeschlichen hat. Dabei weist ein Wert größer/gleich $80 auf einen Fehler hin; die Codes entsprechen den üblichen Definitionen.

Erweiterungen

In den beiden RTP-files sind noch längst nicht alle ACTION!-Befehle eingebaut. Aber Sie können damit auf einen Grundstock zurückgreifen und beliebige Erweiterungen vorhemen. Fehlt Ihnen z.B. der SOUND-Befehl, müssen Sie nur eine Prozedur SOUND(…) anlegen, der Compiler wird dann immer auf Ihren SOUND-Befehl zurückgreifen und den im ROM eingebauten nicht mehr beachten. Natürlich dürfen Sie in SOUND() nur Befehle verwenden, die schon zuvor im Rahmen der RTP definiert wurden.

Ich hoffe, dass die Assembler-Fans nicht verärgert sind, weil es diesmal eine reine ACTION!-Ecke geworden ist. Ich glaube aber, dass sich inzwischen auch viele ML-Freaks mit Sprachen wie ACTION! beschäftigen und die Assemblerecke auch ein Forum für assemblernahe Sprachen sein sollte. Zum Trost: in der nächsten Ausgabe gibt's wieder reichlich Futter für den Assembler.

```
;**************************************
;  ACTION!-Runtime Package TEIL 1
;
;         Filename: RUN1.ACT
;
;PETER FINZEL                    1986
;**************************************

;*****  Multiplication *****
PROC RUDIV2=*()
[ $85 $86 $86 $87 $38 $A9 $00 $E5 $86
$A8 $A9 $00 $E5 $87 $AA $98 $60 ]

PROC Rumlt0=*()
[ $F0 $1B $CA $86 $C1 $AA $F0 $15
$86 $C0 $A9 $00 $A2 $08 $0A $06 $C0
$90 $02 $65 $C1 $CA $D0 $F6 $18
$65 $87 $85 $87 $A5 $86 $A6 $87 $60 ]

PROC Rumlt1=*()
[ $86 $C2 $E0 $00 $10 $03 $20 RUDIV2
$85 $82 $86 $83 $A5 $85 $10 $0E
$AA $45 $C2 $85 $C2 $A5 $84 $20
RUDIV2 $85 $84 $86 $85 $A9 $00 $85
$87 $60 ]

PROC Mult=*()
[ $20 RUMLT1 $A6 $82 $F0 $1B $86 $C0
$A6 $84 $F0 $15 $CA $86 $C1 $A2
$08 $0A $26 $87 $06 $C0 $90 $06
$65 $C1 $90 $02 $E6 $87 $CA $D0 $F0
$85 $86 $A5 $82 $A6 $85 $20 RUMLT0
$A5 $83 $A6 $84 $20 RUMLT0 ]

;***** Division *****
PROC RUDIV=*()
[ $A4 $C2 $10 $03 $4C RUDIV2 $60 ]

PROC Div=*()
[ $20 RUMLT1 $A5 $85 $F0 $27 $A2 $08
$26 $82 $26 $83 $26 $87 $38 $A5
$83 $E5 $84 $A8 $A5 $87 $E5 $85
$90 $04 $85 $87 $84 $83 $CA $D0 $E7
$A5 $82 $2A $A2 $00 $A4 $83 $84 $86
$18 $90 $1D $A2 $10 $26 $82 $26
$83 $2A $B0 $04 $C5 $84 $90 $03
$E5 $84 $38 $CA $D0 $EF $26 $82 $26
$83 $85 $86 $A5 $82 $A6 $83 $A4 $C2
$10 $10 $85 $84 $86 $85 $38 $A9
$00 $E5 $84 $A8 $A9 $00 $E5 $85
$AA $98 $60 ]

;***** Modulo *****
PROC Modulo=*()
[ $20 DIV $A5 $86 $A6 $87 $60 ]
```

```
;***** Links u. Rechtsschieben *****
PROC Rrsh=*()
[ $A4 $84 $F0 $0A $86 $85 $46 $85
$6A $88 $D0 $FA $A6 $85 $60 ]

PROC Rlsh=*()
[ $A4 $84 $F0 $0A $86 $85 $0A $26
$85 $88 $D0 $FA $A6 $85 $60 ]

;***** Parameter-Routine *****
PROC Par=*()
[ $85 $A0 $86 $A1 $84 $A2 $18 $68 $85
$84 $69 $03 $A8 $68 $85 $85 $69
$00 $48 $98 $48 $A0 $01 $B1 $84 $85
$82 $C8 $B1 $84 $85 $83 $C8 $B1
$84 $A8 $B9 $A0 $00 $91 $82 $88
$10 $F8 $A5 $11 $D0 $05 $E6 $11
$6C $0A $00 $60 ]

SET $4E4=Rlsh
SET $4E6=Rrsh
SET $4E8=Mult
SET $4EA=Div
SET $4EC=Modulo
SET $4EE=Par
```

```
;**************************************
;  ACTION!-Runtime Package TEIL 2
;
;         Filename: RUN2.ACT
;
;PETER FINZEL                    1986
;**************************************

;Globale-Variable fuer Fehler etc.
;================================

MODULE
DEFINE Dev="0"
BYTE ARRAY EOF(7)=$5C0
BYTE ioerr

;Hilfsfunktionen fuer IO-Befehle
;==============================

PROC CIOL=*(BYTE chn,cmd,
            CARD Buffer,Length)
[ $85 $A0 $86 $A1 $0A $0A $0A $0A $AA
$A5 $A1 $9D $42 $03 $98 $9D $44
$03 $A5 $A3 $9D $45 $03 $A5 $A4
$9D $48 $03 $A5 $A5 $9D $49 $03 ]

PROC CIO=*()
[ $20 $56 $E4 $A6 $A0 $85 $A0 $C0 $88
$D0 $09 $A9 $01 $9D $C0 $05 $8D
ioerr $60 $A9 $00 $9D $C0 $05
$8C $FF $06 $60 ]

BYTE FUNC CIO5=*(BYTE chn,cmd,data)
[ $85 $A0 $86 $A1 $0A $0A $0A $0A $AA
$A5 $A1 $9D $42 $03 $A9 $00 $9D
$48 $03 $9D $49 $03 $98 $4C CIO ]

PROC SETAUX=*(BYTE chn,aux1,aux2)
[ $86 $A1 $84 $A2 $0A $0A $0A $0A $AA
$A5 $A1 $9D $4A $03 $A5 $A2 $9D
$4B $03 $60 ]

;** OPEN- and CLOSE-Command **

PROC Open(BYTE chn,
          BYTE POINTER fname,
          BYTE aux1,aux2)
BYTE ARRAY fstr(17)
BYTE POINTER bptr
BYTE z

bptr=fname+1
FOR z=0 TO fname^-1
    DO fstr(z)=bptr^  bptr==+1  OD
fstr(z)=$9B
SETAUX(chn,aux1,aux2)
CIOL(chn,3,fstr,0)
RETURN
```

```
PROC Close(BYTE chn)
CIOS(chn,$0C,0)
RETURN

;GET- and PUT-Befehle
;====================

BYTE FUNC GetD(BYTE chn)
RETURN (CIOS(chn,7,0))

PROC Put(Byte chr)
CIOS(device,$0B,chr)
RETURN

PROC PutE()
CIOS(device,$0B,$9B)
RETURN

PROC PutD(BYTE chn,chr)
CIOS(chn,$0B,chr)
RETURN

PROC PutDE(BYTE chn)
CIOS(chn,$0B,$9B)
RETURN

;PRINT-Befehle fuer Strings
;==========================

PROC PrintD(BYTE chn,
            BYTE POINTER buffer)
CIOL(chn,$0B,buffer+1,buffer^)
RETURN

PROC PrintDE(BYTE chn,
             BYTE POINTER buffer)
PrintD(chn,buffer)  PutDE(chn)
RETURN

PROC Print(BYTE POINTER buffer)
PrintD(Dev,buffer)
RETURN

PROC PrintE(BYTE POINTER buffer)
PrintDE(Dev,buffer)
RETURN

;GRAPHICS-Befehl
;===============

PROC Graphics(BYTE Gr)
Close(6)
Open(6,"S:",(Gr&$F0)!$1C,Gr)
RETURN
```