Das Wesentliche zum CPC 464

zusammengefasst von Gerd Raudenbusch

Inhalt

VORWORT 3
1. GRAFIK 4
  1.1. Farben *4*
  1.2. Bildschirm-Maskierung *6*
  1.3 Bildschirm-Speicherzuordnung *8*
  1.4 Gate-Array *11*
  1.5 CRTC Videocontroller HD6845 .... *13*
  1.6 Sprites *17*
2. SOUND 18
  2.1 Der 8255 im CPC *18*
  2.2 Der PSG (AY-3-8912) *20*
3. DATENTRÄGER 22
  3.1. Laden und Speichern *22*
  3.2. Daten über Files *23*
  3.3. Die Datasette *24*
    3.3.1. Blockaufbau 24
    3.3.2. Baudrate 25
    3.3.3. Lese-/Schreibfehler 25
    3.3.4 Kopierschutz-möglichkeiten 26
  3.4. Die DDI1-Floppy *30*
    3.4.1. Directory und Disk-Formate 31
    3.4.2 Lese/Schreibfehler 35
    3.4.3 Gelöschte Files retten 35
    3.4.4 Kopierschutz-möglichkeiten 36
4. INTERN 37
  4.1. BASIC *37*
  4.2 RSX-Befehle *41*
  4.3. Interrupts *42*
  4.4. Daten über den Computer *43*
  4.5. Control-Zeichen *44*
  4.6 Einfache Verschlüsselungstechnicken *45*
  4.7 CP/M *47*
5. ANHANG 52
  5.1. Binär-Arithmetik *52*
  5.2. Register und Flags des Z80 .... *54*
  5.3. Der Stapel *55*
  5.4. Speicheraufteilung *56*
  5.5. RST's *57*
  5.6. Der Z80-Befehlssatz *60*
    5.6.1. Funktionen der Befehle 60
    5.6.2. Die Befehlstabelle 62
  5.7. Firmware und reservierter Speicher *67*
    5.7.1. Allgemeine Vektoren 67
    5.7.2. Die Firmware 74
    5.7.3. Basic-Vektoren 77
  5.8. Hardware-Daten *80*
    5.8.1. Joystick-Port 80
    5.8.2. Druckerport (&EF d. 8255) 80
    5.8.3. Bildschirm-Ausgang 80
    5.8.4. Expansions-Port 81

Vorwort

Ich bekam 1988 einen Schneider CPC464 geschenkt - da war ich gerade 13, und habe mich so für die Maschine begeistert, dass ich Nächte damit verbracht habe, Programme und Spiele zu schreiben, Kopierschutz knacken, usw. Der PC hat mich dann davon weggebracht - man geht ja mit der Zeit und dort war auf ein Mal alles so komplex, dass ich resignierte - ausserdem wollte ich Assembler nicht nochmal neu lernen (den 80x86-Befehlssatz).

Nach zwei Jahren nämlich hatte ich BASIC satt und fing recht schnell an, hinter diese MC-Geschichte zu steigen. Da der einzige Assembler, den ich hatte, 14 Seiten auf Paperware war und ich zu faul zum abtippen war, suchte ich mir die OP-Codes für die Assembler-Befehle immer aus der Tabelle raus und pokte sie mit einem kurzen Programmchen ein. Der positive Nebeneffekt war, dass ich bald mein eigener Assembler/Disassembler war und fast die gesamte Tabelle und Firmware auswendig kannte (teilweise noch kann).

Später habe ich dann durch Zufall einen Assembler in die Hände bekommen und dann auch grössere Programme geschrieben - zum Beispiel einen Disassembler, der mir noch fehlte. Dann ging irgendwann mein PC kaputt - ich hatte kein Geld, ne Menge Zeit und so fing ich an - nun ein paar Jahre älter - mir all das Wissen zu ergründen, dass ich früher immer gerne gehabt hätte, z.B. um Speedlock-Loader zu machen oder um den GANZEN Bildschirm zu verwenden.

Wem dieses Wissen aus Nostalgie-Gründen oder zum kurz mal nachschlagen noch was nützt, der soll dieses Skript gespannt lesen - leider kann ich nur mit 464-spezifischen Informationen dienen, aber ein guter Emulator (ich empfehle den CPCEMU v1.5) macht ja fast alles mit. Nur die ganz harten Tricks kriegt auch der nicht hin, da wird man wohl doch die alte CPC-Kiste (falls man sie noch hat) wieder zum laufen bringen und Firmware-spezifischen Dinge ändern müssen...

Viel Spass also - und für neue Informationen bin ich immer(noch) offen :)) Hinweise, Tricks, Tips, Hints bitte senden an elysis@gmx.net - ich werde mich bemühen, das Skript auf dem Laufenden zu halten und es durch das Internet jedem zugänglich zu machen.

Bitte beachtet, dass ich als Autor keinerlei Haftung für irgendwelche Schäden oder Datenverluste übernehme, die in irgendeiner Weise auf mich oder auf Informationen dieses Skripts zurückzuführen sind. Die Benutzung ist also auf eigene Gefahr und alle Angaben sind ohne Gewähr.

1. Grafik

1.1. Farben

Die Anzahl der gleichzeitig verwendbaren Farben (Farbstifte) ist von dem Bildschirmmodus abhängig :

MODE 0 : 16 Farben
MODE 1 : 04 Farben
MODE 2 : 02 Farben (weiss ja wahrscheinlich jeder)

und diese Farben sind wiefolgt den Farbwerten zugeordnet :

00 schwarz	**14 pastellblau**
01 blau	**15 orange**
02 hellblau	**16 rosa**
03 rot	**17 pastellmagenta**
04 magenta	**18 hellgrün**
05 hellviolett	**19 seegrün**
06 hellrot	**20 helles blaugrün**
07 purpur	**21 limonengrün**
08 hellmagenta	**22 pastellgrün**
09 grün	**23 passtellblaugrün**
10 blaugrün	**24 hellgelb**
11 himmelblau	**25 pastellgelb**
12 gelb	**26 weiss**
13 grau	

Diese Farben können den zur Verfügung stehenden Farbstiften zugeordnet werden.

In BASIC :

```
INK <Farbstift>,<Farbwert1>,<Farbwert2>
```

In MC/ASM :

```
LD a, <farbstift>             3Exx
LD bc,<farbwert2><farbwert1>  01yyxx
CALL &BC32                    CD32BC
RET                           C9
```

Soweit sogut. Intern existiert ein Interrupt, der die Farbwerte ständig an den Grafik-Chip des CPC (Gate Array) schickt und sie somit auffrischt. Wo ist dieser Interrupt? An der Speicherstelle &B1FE steht der Eventblock, dazu gehört der Event-Counter an der Stelle &B202. Setzt man ihn negativ (Wert 128-254), dann werden die Farben nicht aufgefrischt, da können noch soviele INK-Kommandos kommen.

Wo im Speicher sind diese Farbwerte nun gespeichert ?
Ab der Stelle &B1D7 in folgender Weise :

Farbstift	Farbwert2	Farbwert1	Farbstift	Farbwert2	Farbwert1
Border	&B1D9	&B1EA	08	&B1E2	&B1F3
00	&B1DA	&B1EB	09	&B1E3	&B1F4
01	&B1DB	&B1EC	10	&B1E4	&B1F5
02	&B1DC	&B1ED	11	&B1E5	&B1F6
03	&B1DD	&B1EE	12	&B1E6	&B1F7
04	&B1DE	&B1EF	13	&B1E7	&B1F8
05	&B1DF	&B1F0	14	&B1E8	&B1F9
06	&B1E0	&B1F1	15	&B1E9	&B1FA
07	&B1E1	&B1F2	Blink-Periode	&B1D7	&B1D8

Liest man dort den Speicher aus, stösst man auf von den Farbwerten verschiedene Zahlen. Woran liegt das? Diese Zuordnung wurde vorgenommen, damit die Farbpalette des CPC der Norm entspricht (Man lasse sich ersten 8 Farben auf den Bildschirm malen und vergleiche mit Fernseh-Testbildern). Das heisst also, dass die Farben intern ganz andere Werte haben, wir sprechen hier von den Hardware-Farbwerten. Wie genau sind diese nun zueinander zugeordnet ?

Software	Hardware	Software		Hardware
00	20	16		07
01	04	17		15
02	21	18		18
03	28	19		02
04	24	20		19
05	29	21		26
06	12	22		25
07	05	23		27
08	13	24		10
09	22	25		03
10	06	26		11
11	23	-- 27	-------	01 --
12	30	28		08
13	00	29	wdh.	09
14	31	30		16
15	14	-- 31	-------	17 --

Die Farbpalette lässt sich also speichern durch :

SAVE"<name>",b,&B1D7,&28

und sie stellt sich nach dem Laden auch wieder selbständig ein, sofern der Ticker-Counter nicht negativ gemacht wurde.

Was bringt das ? Naja, man könnte zum Beispiel ein Mini-MC (von wegen LDIR und so) am Ende eines Bildes unterbringen (Ab &FFD0, da dieser Bereich sowieso nicht mehr sichtbar ist), und die nachfolgenden Farbwerte damit direkt nach &B1D9 verschieben. Dann würde man sich die dämlichen INK- bzw. DATA-Kommandos sparen. Aber die meisten werden Bilder sowieso komprimieren, also, was solls...

Dem Bildschirm ist regulär der Speicherbereich &C000 bis &FFFF zugeordnet, ein Bildschirm kann also mit **SAVE"<name>",b,&C000,&4000** gespeichert und mit **LOAD"<name>",&C000** wieder geladen werden. Wie man beim Laden von Screens oder beim beschreiben gemerkt haben wird, ist der Bildschirmaufbau beim CPC etwas verworren. Nicht nur, dass er nicht Zeile für Zeile aufbaut, sondern 8 Zeilen weiterspringt, nein, da gibts auch noch ein Problem mit den Farben - die Bytes sind nämlich maskiert und in MODE 0 passen in ein Byte 2 Bildpunkte, während in MODE 2 genau 8 Punkte reinpassen. Was soll das alles ?

1.2. Bildschirm-Maskierung

Dem Bildschirmbereich ist eine feste Länge von &4000 zugeordnet. Die Grössen PIXELANZAHL und FARBANZAHL müssen sich dabei für alle Bildschirm-Modi SO ergänzen, dass diese Länge eingehalten wird. Dies hat zur Folge, dass in MODE 0 die Auflösung geringer ist. Daran lässt sich auch mit allen Tricks nichts ändern - leider. Es gibt zwar Routinen, die durch geschickten Umgang mit Interrupts mehr Farben zur Verfügung stellen, dies bringt aber wiederum viele Emulatoren ins Schwitzen.

Wie funktioniert nun diese Maskierung ? Nehmen wir zunächst den Bildschirm-Modus MODE 2. Hier ist alles ganz einfach - Ein Punkt wird einem Bit zugeordnet. Da ein Byte bekanntlich 8 Bit hat, kann es also 8 Bildpunkte speichern, wobei eine 0 für die Hintergrund-Farbe, eine 1 für die Vordergrund-Farbe steht.

Wie sieht das Ganze in MODE 1 aus ? Plotten wir also einmal einen Punkt in die linke, oberste Ecke mit **PLOT 0,399,1** und lesen das erste Screen-Byte mit PRINT PEEK(&C000). Wir erhalten den Wert 128 - in Bits zerlegt 10000000. Anscheinend hat sich nichts geändert. Ändern wir nun aber die Farbe (**PLOT 0,399,2**) so erhalten wir den Wert 8 - also 00001000. Für die Farbe 3 erhalten wir den Wert 10001000. Das Byte wird also in zwei Hälften (low-nibble und high-nibble) geteilt, wobei die Position ausschlaggebend für die Farbe ist :

MODE 2 : Tatsächliche Pixel+Farbe	**Screenbyte**
1 0 0 0 0 0 0 0	**1 0 0 0 0 0 0 0**
0 1 0 0 0 0 0 0	**0 1 0 0 0 0 0 0**
.	.
.	.

MODE 1 :			
	1 0 0 0	**1 0 0 0**	**0 0 0 0**
	2 0 0 0	**0 0 0 0**	**1 0 0 0**
	3 0 0 0	**1 0 0 0**	**1 0 0 0**
		Farbbit0	**Farbbit1**

Bei MODE 0 wird sogar in sog. 2-Tupel (paarweise Bits) unterteilt :

```
MODE 0 :                 1 0              1 0     0 0     0 0     0 0
                         2 0              0 0     0 0     1 0     0 0
                   ----- 4 0              0 0     1 0     0 0     0 0
                   !     8 0              0 0     0 0     0 0     1 0
                   !
                   !                    F.Bit0  F.Bit2  F.Bit1  F.Bit3
                   !
                   !------> Als Beispiel in MODE 0:
                            PLOT 0,399, 4 \
                            PLOT 4,399, 0 /  PEEK(&C000)=32 = &X 00 10 00 00
```

Ab &B1CF bis &B1D6 stehen diese maskierenden Bytes. In MODE 2 sind es 8, in MODE 1 nur 4 und in MODE 0 sind es 2 :

MODE 0 : AA 55
MODE 1 : 88 44 22 11
MODE 1 : 80 40 20 10 08 04 02 01

Ist die Logik erkennbar? Nein? Dann erläutern wir den Algorithmus anhand eines Beispiels : Es soll die erste Pixelzeile eines "K" auf den Bildschirm gepoked werden.

In Mode 2 ist das kein Problem : POKE &C000,&X11100110
In Mode 1 nehmen wir für den Farbstift 1 die Maske &F0
den Farbstift 2 die Maske &0F
den Farbstift 3 die Maske &FF,

und poken &C000,&X11100110 AND <maske>
&C001,&X01101110 AND <maske> (hier das Byte um 4 n. links rotiert)

In MODE 0 rotieren wir das Byte jeweils nur noch um 2.
Daraus ergeben sich die oben genannten Masken für die Modis.

MODE 1:

```
INK 0 - ---X = 0000 0000 (0)         INK 2 - ---X = 0000 0001 (1)
INK 0 - --XX = 0000 0000 (0)         INK 2 - --XX = 0000 0011 (3)
INK 0 - -XXX = 0000 0000 (0)         INK 2 - -XXX = 0000 0111 (7)
INK 0 - XXXX = 0000 0000 (0)         INK 2 - XXXX = 0000 1111 (15)   (&0F)
INK 1 - ---X = 0001 0000 (16)        INK 3 - ---X = 0001 0001 (17)
INK 1 - --XX = 0011 0000 (48)        INK 3 - --XX = 0011 0011 (51)
INK 1 - -XXX = 0111 0000 (112)       INK 3 - -XXX = 0111 0111 (119)
INK 1 - XXXX = 1111 0000 (240) (&F0) INK 3 - XXXX = 1111 1111 (255)  (&FF)
```

Mit dem Firmware-Call &BD1C kann man die Bildschirm-Maskierung wechseln, ohne den Bildschirm zu löschen. An der Speicherstelle &B1C8 ist die aktuelle Bildschirm-Mode-Nummer gespeichert. Mit **POKE &B1C8,<mode>** lässt sich also auch die Bildschirm-Mode wechseln, ohne den Bildschirm zu löschen. Also kann man von einer Bildschirm-Mode mit höherer Auflösung auch in eine Mode mit niedrigerer Auflösung wechseln, ohne den Bildschirm zu löschen, indem man einfach die Maskierung manipuliert :

Von Mode 1 nach Mode 0 : **POKE &B1C8,0:POKE &B1CF,&C0:POKE &B1D0,&30**
Von Mode 2 nach Mode 1 : **POKE &b1C8,1:POKE &B1CF,&C0:POKE &B1D0,&30:**
POKE &B1D1,&0C:POKE &B1D2,&03
Von Mode 2 nach Mode 0 : **POKE &B1C8,0:POKE &B1CF,&F0:POKE &B1D0,&0F**

Allerdings lässt sich der Farbstift jetzt nicht mehr ohne weiteres wechseln.

1.3 Bildschirm-Speicherzuordnung

Kommen wir gleich zur Sache.

```
&C000 ........ &C04F \
&C800 ........ &C84F  \
&D000 ........ &D04F   \
&D800 ........ &D84F    \  Erstes Textzeichen (Textpos. 1,1) - (8 Grafik-Zeilen)
&E000 ........ &E04F    /
&E800 ........ &E84F   /
&F000 ........ &F04F  /
&F800 ........ &F84F /

&C050 ........ &C0FF
&C850 ........ &C8FF
&D050 ........ &D0FF
         .
         .
         .
```

das hangelt sich so fort bis &FFD0.

Eine Grafik-Zeile ist also immer &50 lang, und in die nächste Zeile kommen wir, indem wir &800 addieren. Dies können wir solange tun, bis wir eine Adresse ausserhalb des Screenbereichs erhalten, dann müssen wir 8 * &800 abziehen (eine Textzeile) und eine die Länge einer Grafikzeile addieren (&50) und wiederum 1 abziehen. Das ist die Vorgehensweise.

Wie geht man mit so einem besssssssscheidenen Bildschirmaufbau um? Als Beispiel soll folgendes kurze MC dienen, welches den Bildschirm um eine Pixelzeile nach oben scrollt :

```
         ld hl,&C000   ; HL=Bildschirmadresse
  @loop  push hl       ; Zieladr. auf Stapel
         ld de,&800    ; Quelladresse ausrechnen
         add hl,de     ; (+&800)
         call @check   ; grösser &FFFF? eventuelle Korrektur
         pop de        ; DE=Zieladresse (vom Stapel)
         ld bc,&50     ; Länge=&50
         push de       ; Zieladresse sichern (Stapel)
         ldir          ; Zeile kopieren
         pop hl        ; Zieladresse holen (wird zur neuen Startadr.)
         ld de,&800    ; nächste
         add hl,de     ; Zeilenadresse
         call @check   ; berechnen (s.o.)
         ld a,h        ; \
         cp &FF        ; / HI-Byte  \
         jr nz,@loop   ; \           HL=&FF80?
         ld a,l        ; / Lo-Byte  /
         cp &80        ;
         jr nz,@loop   ; nein? Weiter
         ret           ; ja?   Ende

  @check ld a,h        ; Unterroutine falls HL>&FFFF
         cp &c0        ; H <&C0? (&FFFF+&0001 = &0000 <&C000)
         ret nc        ; Nein? -> Fertig
         ld de,&3FAF   ; ansonsten
         sbc hl,de     ; HL=HL-&3FAF (-&4000 (=8*&800) + &50 - 1)
         ret           ; Fertig
```

Der Zeilenvorschub lässt sich noch etwas abkürzen, WENN der Bildschirm denn wirklich bei &C000 anfängt (man kann den Bildschirm nämlich auch nach &4000 legen) :

```
         ld hl,&C000 ; HL=Bildschirmstartadr.
         ld b,&C8    ; = Bildschirmlänge (&3E80 / Zeilenlänge &50)
   @loop push bc     ; (merken)
         push hl     ; (merken)
          .
          .
       ( - Hier steht der Hauptteil der Bildschirmbearbeitung - )
          .
          .
         pop hl      ; (HL wieder holen)
         ld bc,&800  ; Eine Zeile
         add hl,bc   ; nach unten
         jr nc,@ok   ; hl < 0?   (funktioniert nur bei &C000 als Startadr.)
         ld bc,&C050 ; (=Bildschirmadr. + Zeilenlänge &50)
         add hl,bc   ; Adresse korrigieren
   @ok   pop bc      ; Counter vom Stapel holen
         djnz @loop  ; fertig? Nein-> Weiter
         ret         ;           Ja-> Ende
```

Folgender Hauptteil scrollt den Bildschirm um 1 Byte nach links :

```
         ld d,h      ; \ DE=HL
         ld e,l      ; /
         inc hl      ; Zieladresse berechnen
         ld bc,&4F   ; Länge einer Zeile
         ldir        ; Transfer
```

Folgender Hauptteil scrollt den Bildschirm um 1 Byte nach rechts :

```
         ld bc,&4f   ; Zieladr. = Quelladr. + &4f
         add hl,bc   ;
         ld d,h      ; \
         ld e,l      ;   de=hl - 1
         dec hl      ; /
         lddr        ; Transfer
         inc hl      ;
         ld (hl),0   ; und Rest mit 0 überspielen (sonst gibt's Streifen)
```

Etwas komplizierter wird es schon, wenn wir den Zeilenvorschub für (mit OUT) vergrösserte/verkleinerte Bildschirme verwenden (siehe Kapitel 1.3) Wer schon mit den Bildschirm-OUTs rumgefummelt hat (&BC00, &BD00) weiss, dass man damit den Bildschirm vergrössern und verkleinern kann. Manche Spiele (z.B. Cybernoid) arbeiten mit einem verkleinerten Schirm, sodass er im Normalformat verzogen erscheint. Wie berechnet man also für solche Screens den Zeilenvorschub? Folgende Routine behandelt nun Hi- und Lowbyte getrennt und kommt daher mit jeder Bildschirmgrösse zurecht :

```
  @calc  ld a,h      ; \
         add a,8     ;  HL=HL+&800
         ld h,a      ; /
         and &38     ; Bit 6 gesetzt?
         ret nz      ; nein? fertig.
         ld a,h      ; ja ?
         sub &40     ; hl=hl-&4000 (=8*&800)
         ld h,a      ;
         ld a,l      ; \
         add a,<lng> ;  HL=HL+<Zeilenlänge in Bytes>
         ld l,a      ; /
         ret nc      ; Übertrag ? Nein -> Fertig.
         inc h       ;              Ja -> Hi-Byte bearbeiten
         ld a,h      ;
         and 7       ;  >8 ?
         ret nz      ;  Nein - > Fertig, ansonsten
         ld a,h      ; \
         sub 8       ;  &800 wieder abziehen
         ld h,a      ; /
         ret         ;
```

Um nun richtige Macht über den Bildschirm zu gewinnen, ist es notwendig, sich mit der CRTC und dem Gate-Array auseinanderzusetzen. Noch ein Hinweis zu OUTs in Assembler : Dort enthält das B-Register immer die Portadresse, die eigentlich nur ein Byte benötigt.

Die Befehle **OUT &BC00,1** und **OUT &BCFF,1** in BASIC bewirken genau das gleiche, das Lowbyte spielt also keine Rolle. Darüberhinaus ist der Assembler-Befehl **out (C),C** etwas verwirrend, er schreibt den Wert des C Registers in den Port des B-Registers und müsste eigentlich OUT (B),C heissen. Meistens lädt man in das BC-Register gleich PORT und Wert und benutzt dann den Befehl **out (c),c**.

Die Assembler-Zeilen

LD bc,&BC02
out (c),c

entsprechen der Basic-Zeile

OUT &BC00,2

1.4 Gate-Array

Das Gate-Array erzeugt alle nötigen Frequenzen, steuert Zugriffe auf den Speicher, speichert Farbwerte und erzeugt die Interrupt-Impulse. Die Register des Gate-Arrays sind nur beschreibbar und werden wiefolgt angesprochen : Mit Bit 6 und 7 wird das Register ausgewählt, mit Bit 0 bis 4 der Wert übergeben. Bit 5 bleibt unbenutzt.

```
  Bit   7  6  5  4  3  2  1  0

                 \--Wert---/

        0  0  -   FN-Register
        0  1  -   FW-Register
        1  0  -   MF-Register
        1  1  -   unbenutzt (?)
```

```
FN-Register : Farbstift (unabhängig von der Screenmode), 16 = Border
FW-Register : Farbwert (hardw.) an FN-Reg.
MF Register : Bit 5 : -
              Bit 4 : VSync-Zähler zurücksetzen (Framefly)
              Bit 3 : ROM &C000 - &FFFF schalten ( 0 = an, 1 = aus)
              Bit 2 : ROM &0000 - &3FFF schalten ( 0 = an, 1 = aus)
              Bit 1 : 0  0  0  0
              Bit 0 : 0  1  0  1

                      !  !  !  !
                      !  !  !  !-- Mode 0 ohne blinken
                      !  !  !
                      !  !  !----- Mode 0
                      !  !
                      !  !-------- Mode 1
                      !
                      !----------- Mode 2
```

Beispiel für FN/FW-Register :
Mit **OUT &7F00,&X00010000 : OUT &7F00,64+11** ist der Border bis zum nächsten Refresh weiss. (Der lässt sich ja mit **SPEED INK 255,255** verlängern oder mit **POKE &B202,128** abstellen)

Beispielprogramm Streifen :

Folgendes Programm bringt Streifen horizontal über den Bildschirm (auch durch den Border). Diese Technik wird oft in Demos verwendet. Wichtig ist der Firmware-Call &BD19, der dafür sorgt, dass der Übertrag vom Speicher zum Bildschirm immer an der gleichen Stelle beginnt. Ohne ihn laufen die Streifen durch, als wenn das VSync-Signal des Bildschirms verstellt wäre. Hier wird die Speichergrösse &FF benutzt. Die Bytes im Speicher müssen den Anforderungen des FW-Registers des Gate-Array entsprechen, d.h. Bit 5+7 müssen 0, Bit 6 muss 1 sein und die Farbwerde (Bit 0 bis Bit 4) sind Hardware-Farbwerte. Es empfiehlt sich also, den Speicher ab <adr> bis <adr>+&FF mit der Farbe 0 zu füllen (softw. 0 = hardw. 20, siehe Tabelle) = &X010(10100) = &54 und dann die gewünschten Streifen reinzupoken. Die Routine ist sehr schnell und durch ein Rotieren des Speicherbereichs oder eine Sinus-Berechnung kann man damit schöne Effekte erzielen.

```
@main   call &BD19              ; Synchronisieren
        di                      ; Andere Interrupts ausschalten
        call @show              ; Routine soll laufen
        call &bb09              ; Solange die ESC-Taste
        cp 252                  ; nicht
        ret z                   ; gedrückt
        jr @main                ; wird.

@show   ld hl,<adr>             ; <adr> = Speicherbeginn der Farbwerte
        ld bc,&7F10             ; b = &7F (Gate Array);c=&10 (FN-Reg. Border)
        ld de,&20FF             ; d = &20 (FN-Reg. PEN 0);e = Bereichslänge

@loop   ld a,(hl)               ; Farbwert aus Speicher holen
        out (c),c               ; \
        out (c),a               ; / Border bearbeiten
        out (c),d               ; \
        out (c),a               ; / PEN 0 bearbeiten

        ld a,<Streifendicke>    ; \ Verzögerung :
@delay  dec a                   ;   a entspricht der Streifendicke
        jr nz,@delay            ; /   (ungefähr. 1 / Speicherbereich)
        inc hl                  ; nächtes Byte holen
        dec e                   ; am Ende ?
        jr nz,@loop             ; Nein -> Weiter
        ret                     ;   ja -> Fertig.
```

1.5 CRTC Videocontroller HD6845

Der CRTC ist hauptsächlich für die Erzeugung des Bildes verantwortlich.
Seine Registerbelegung sieht folgendermassen aus :

Reg.	Status	Normwerte	Name	Funktion + Wirkung
AR	-/W	0-17	Adress Register	Wählt eines der nachstehenden Reg.
R00	-/W	63	Horizontal Total	Länge d. Bildschirmzeile + Rand + Strahlenrücklauf = 1.5 * Anz. d. dargestellten Zeichen. Wert ausserhalb 46-255 führt z. Absturz.
R01	-/W	40	Horiz. Displayed	Rechter Rand. Angabe IMMER in MODE 1-Textposition. Wert <R0 ist wirkungslos.
R02	-/W	47	Horiz. Sync. Pos.	Bestimmt den Zeitpunkt des HSync-Impulses. Monitorbild verschiebt sich <47 n. rechts, >47 n. links
R03	-/W	04	Sync. Width	Bit 0-4 bestimmen die Breite des HSync- und VSync-Impulses. Wert ausserhalb 1-15 : Absturz (Farbe?) ausserhalb 17-31: ----- " --------
R04	-/W	38	Vertical Total	Rasterzeilen pro Bild (Bildwdh.-frequenz). Wertebereich 0-127, danach Wiederholung.
R05	-/W	00	Vert. Tot. Adjust	Feinabgleich v. R4
R06	-/W	25	Vert. Displayed	Anzahl d. darzust. Zeichen. Wertebereich 0-127, <R4
R07	-/W	30	Vert. Sync. Pos.	Bestimmt den Zeitpunkt des VSync-Inpulses. Monitorbild verschiebt sich <30 n. unten, >30 n. oben. Wertebereich 0-127
R08	-/W	??	Interlace	Darstellung mit/ohne Zeilensprung-Verfahren. Wertebereich 0-3
R09	-/W	07	Max. Raster Adr.	Bestimmt die Anzahl der Raster-Zeilen für Zeichen. Wertebereich 0-31
R10	-/W	??	Cursor Start Raster	"Cursor" bezieht sich auf ein Strobe-Signal (für Lightpen). Bit 6 + 5 Cursormodus : ---------------------- 0 0 nicht blinkend 0 1 kein Cursor 1 0 blinkend 3x / sec. 1 1 blinkend 1.5x / sec.

Reg.	Status	Normwerte	Name	Funktion + Wirkung
R11	-/W	??	Cursor End Raster	Bestimmt, auf welcher Rasterzeile der "Cursor" endet.
R12	R/W	&C0	Startadr. High	Adresse d. Bildschirms (Hi-Byte) Momentan sichtbarer Bereich bleibt vorhanden. Beim Lesen sind Bit 6+7 immer 0.
R13	R/W	&00	Startadr. Low	Lowbyte zu R12
R14	R/W	??	Cursor High	Highbyte der aktuell. "Cursor"-Pos.
R15	R/W	??	Cursor Low	Lowbyte der aktuell. "Cursor"-Pos.
R16	R/-	--	Strobe Hi	Enthält nach positivem Strobeimpuls das Highbyte der zur Impulszeit aktuellen Bildschirmadr. Bit 6+7 immer 0.
R17	R/-	--	Strobe Lo	Lowbyte zu R16

Beispiel : **Mit OUT &BC00,01:OUT &BD00,20** wird die Bildschirmgrösse (rechts-links) halbiert. Dadurch kommt es zu einer Verschiebung, bzw. Verzerrung der Bildschirmzeilen, die wir mit dem MC im vorangegangenen Kapitel behandeln können.

Noch ein Hinweis zu R12/R13 der CRTC :
Man kann ja den Bildschirm-Bereich mit dem Firmware-Call &BC06 bzw. &BC08 verschieben. Also der BASIC-Befehl **CALL &BC06,&40** (in ASM **:LD a,&40, JP &BC08**) bewirkt, dass der Speicherbereich des Bildschirms &4000 bis &8000 ist. Dabei wird gleich in diesen Speicherbereich übergeblendet, d.h. es wird sofort sichtbar, was sich dort befindet. Die Manipulation des R12/R13 der CRTC hingegen bewirkt, dass der aktuelle Bildschirminhalt sichtbar bleibt, aber alle neuen Optionen sich nun auf den neuen Bildschirm-Bereich beziehen, wodurch man den Eindruck gewinnen könnte, der Computer sei abgestürzt, weil man keinen Cursor mehr sieht.

Wenn wir den Bildschirm-Bereich nun vergrössern wollen, also in R01 des CRTC einen Wert >40 schreiben, müssen wir mit Bedauern feststellen, dass die Inhalte der gewonnenen Spalte nun doppelt erscheinen, da der Screen-Speicher nunmal nicht grösser als &4000 wird. Beispielsweise entfernen die OUT-Befehle :

```
OUT &BC00,1:OUT &BD00,50
OUT &BC00,2:OUT &BD00,55
OUT &BC00,6:OUT &BD00,39
```

den Border, aber das Problem des doppelten oder sogar dreifachen Erscheinens mancher Zeichen bleibt bestehen. Ich möchte nun ein Programm vorstellen, mit dem es tatsächlich möglich ist, den GANZEN Bildschirm zu nutzen – der Border wird dabei vollständig gekillt !

Wie geht sowas?

Zunächst werden per Interrupt zwei Speicherbereiche sichtbar gemacht, nämlich &0000-&4000 und &C000-&FFFF. Interessant dabei ist der CALL &B939, mit dem sich die Bereiche optisch vertauschen lassen und es sind sogar zwei verschiedene Bildschirmmodi auf einmal möglich. Dieses Programm ist sehr auf Timing angewiesen, deshalb wird es da wohl Schwierigkeiten mit einem Emulator geben.

Im Bereich &0000-&003F stehen die RSTs. Wenn man sie im Eingabe-Modus löscht, stürzt der Rechner ab, weil er diese benutzt. Sie liegen aber andererseits im Bildbereich und müssen deswegen solange irgendwo zwischengespeichert und am Ende wieder an den alten Platz verschoben werden. Zum Programm, dass diese Unmöglichkeit möglich macht :

```
        jp @on
        jp @off

 @on    ld hl,&0000                ; Einschalten : Die RSTs
        ld de,&BF00                ; sichern
        ld bc,&003F                ; (nach &BF00
        lddr                       ; verschieben)
        ld de,@cont1               ; Ersten Block
        call @poke                 ; an den Interrupt-Verteiler
        ld hl,@ende                ; Nun
        ld de,@start               ; Die Interrupt
        ld bc,&8100                ; einhängen HL=Startadr.,
        jp &BCE0                   ; DE=Eventblock, BC=Speicherkonfig.

 @off   ld hl,@ende                ; Ausschalten :
        ld de,@start               ; Zunächst den Interrupt
        call &bce6                 ; aushängen,
        ld hl,&BF00                ; nun die RSTs
        ld de,&0000                ; wieder
        ld bc,&003F                ; zurückkopieren
        lddr                       ;
        jp @norm                   ; und Normwerte der CRTC einstellen.

 @poke  ld hl,@jump                ; Dies ist der Interrupt-Verteiler,
        ld a,e                     ; der dafür sorgt, dass die Blöcke
        ld (hl),a                  ; @cont1 bis @cont6 nacheinander
        inc hl                     ; aufgerufen werden, indem einfach
        ld a,d                     ; die JUMP-Adresse
        ld (hl),a                  ; verändert wird.
        ret

 @start db &c3                     ; Hier ist der JUMP auf den der
 @jump  dw &0000                   ; Interrupt angesetzt ist. Es werden
                                   ; die Adressen der Blöcke @cont1 bis
                                   ; @cont6 nacheinander eingefügt.

 @cont1 ld de,@cont2               ; Jump verbiegen auf nächsten Block,
        call @poke                 ; für nächsten Interrupt-Aufruf
        ld bc,&BC06 : out (c),c    ; Jetzt die CRTC manipulieren.
        ld bc,&bd14 : out (c),c    ; Die Wahl der Bildschirmmode kann
        ld bc,&BC0D : out (c),c    ; hier mit :
        ld bc,&BD00 : out (c),c    ; exx : res/set 0/1,c : out (c),c : exx
        ld bc,&BC0C : out (c),c    ; bestimmt werden (siehe Gate-Array)
        ld bc,&bd00 : out (c),c    ;
        ret

 @cont2 ld de,@cont3               ; Jeweils zwei Blöcke
        call @poke                 ; (@cont1+@cont2 bzw. @cont4+@cont5
        ld bc,&BC04 : out (c),c    ; sind für die zwei Bildschirmbereiche
        ld bc,&BD13 : out (c),c    ; verantwortlich.
        ld bc,&BC07 : out (c),c
        ld bc,&BD24 : out (c),c
        ret
```

```
@cont3  ld de,@cont4                 ; @cont3 und @cont6
        jp @poke                     ; korrigieren das Timing.

@cont4  ld de,@cont5                 ;
        call @poke                   ; Diese zwei Blocks sind
        ld bc,&BC0D : out (c),c      ; für die Anzeige des zweiten
        ld bc,&BD00 : out (c),c      ; Speicherbereichs zuständig.
        ld bc,&BC0C : out (c),c      ;
        ld bc,&BD30 : out (c),c      ; Auch hier kann wieder
        ret                          ; eine MODE-Bestimmung eingebaut werden.

@cont5  ld de,@cont6                 ;
        call @poke                   ;
        ld bc,&BC04 : out (c),c      ;
        ld bc,&BD12 : out (c),c      ;
        ld bc,&BC06 : out (c),c      ;
        ld bc,&BD10 : out (c),c      ;
        ld bc,&BC07 : out (c),c      ;
        ld bc,&BD10 : out (c),c      ;
        ret

@cont6  ld de,@cont1                 ; Timing
        jp @poke                     ; korrigieren (Bereich zw. unten+oben)

@norm   ld bc,&BC02 : out (c),c      ; Hier werden die Normwerte
        ld bc,&BD2F : out (c),c      ; an die CRTC zurückgeschrieben.
        ld bc,&BC04 : out (c),c      ;
        ld bc,&BD26 : out (c),c      ;
        ld bc,&BC06 : out (c),c      ;
        ld bc,&BD19 : out (c),c      ;
        ld bc,&BC07 : out (c),c      ;
        ld bc,&BD1E : out (c),c      ;
        ret                          ;

@ende   db 0                         ; Platz für Event-Block d. Interrupts.
```

1.6 Sprites

Eigentlich sind Sprites keine grosse Sache - wie wir ja wissen, sind Sprites die Grundlage eines jeden Computerspiels. Sprites sind nämlich Speicherblöcke, die Grafikdaten (Spielfiguren, usw.) enthalten und die irgendwie auf dem Bildschirm bewegt, also herumkopiert werden. Dienlich hierzu ist der Firmware-Call &BC1D, der zu den Pixelkoordinaten als Eingangsparameter die Bildschirmadresse liefert. Zu beachten ist, dass die Koordinaten dafür geteilt werden müssen :

320,200 = 320,100 in MODE 2
320,200 = 160,100 in MODE 1
320,200 = 80,100 in MODE 0

Der Befehl SRL rotiert ein Byte nach rechts und teilt es somit durch 2, da wir jedoch 16 Bit rotieren wollen, müssen wir auf den Übertrag achten :

```
srl h     ; Highbyte teilen
push af   ; Carry retten (falls Übertrag)
srl l     ; Lowbyte teilen
pop af    ; Carry holen
ret nc    ; Kein Übertrag? Ja->Fertig
set 7,l   ; Nein -> Das Bit das rausrotiert wurde (ins Carry)
ret       ; wieder rein und Ende.
```

Dann muss der Sprite zeilenweise auf den Bildschirm. In b also die Länge für Y, bc merken, Zeile mit LDIR kopieren. Der Zeilenvorschub kann wie im Kapitel 1.2 erwähnt oder mit dem Firmware-Call &BC26 geschehen. Dann bc wieder holen (pop) und ein DJNZ - fertig.

2. Sound

Um Sound professionell realisieren zu können, ist es wichtig, über den Soundchip, auch PSG (Programable Sound Generator) genannt, bescheid zu wissen. Da dieser jedoch wiederum durch den Portbaustein 8255 gesteuert wird, muss man auch über diesen bescheid wissen. Dieser spielt übrigens auch bei dem Schreiben und Lesen von Datasette und bei der Tastaturabfrage eine Rolle.

2.1 Der 8255 im CPC

Die von der CPU kommenden Daten können auf drei verschiedene 8-Bit-Ports gelegt werden, bzw. umgekehrt. Port A+B können nur entweder als Ein- oder Ausgang benutzt werden, während Port C sich spalten lässt. Alle drei Ports arbeiten im CPC in der Betriebsmode 0 des 8255, d.h. Einfache Ein/Ausgabe. Betriebsmode 1 würde Port C spalten (Steuersignale f. Port A und B) und Betriebsmode 2 wäre ein getasteter bidirektionaler Betrieb. Diese Modi können wir aber gleich wieder vergessen, der CPC arbeitet in Betriebsmode 0.

```
Registerbeschreibung des 8255 :
Bit 7 ist Kennzeichenbit. Ist es gesetzt, so haben Bis 0-6 folgende Bedeutung:

    Bit 6,5 wahlen Modus Gruppe A

    0 0 = Betriebsmode 0                        \
    0 1 = Betriebsmode 1                         (bedeutungslos) f. CPC
    1 x = Betriebsmode 2, Bit 5 ohne Bedeutung  /


Bit 4 (Port A)         : 1 = Eingang, 0 = Ausgang
Bit 3 (Port C Bit 4-7) : 1 = Eingang, 0 = Ausgang
Bit 2 Modus Gruppe B   : 1 = Betriebsmode 0, 0 = Mode 1 (bedeutungslos f. CPC)
Bit 1 (Port B)         : 1 = Eingang, 0 = Ausgang
Bit 0 (Port C Bit 0-3) : 1 = Eingang, 0 = Ausgang

Ist Bit 7 dagegen 0, bedeuten die anderen Bits folgendes :

Bit 6-4 : unbenutzt
Bit 3-1 : Bitauswahl Port C


    000 = Bit 0 \
    001 = Bit 1  \
    010 = Bit 2   \
    011 = Bit 3    wobei Bit 0 (s.oben) zum ausgewählten Bit des Port C
    100 = Bit 4    gesendet wird.
    101 = Bit 5   /
    110 = Bit 6  /
    111 = Bit 7 /
```

Wie der Verdrahtung im CPC zu entnehmen ist, ergeben sich als Port-Adressen :

```
               A15 A14 A13 A12 A11  A10 A09 A08

    &F400       1   1   1   1   0    1   0   0   = Port A Register
    &F500       1   1   1   1   0    1   0   1   = Port B Register
    &F600       1   1   1   1   0    1   1   0   = Port C Register
    &F700       1   1   1   1   0    1   1   1   = Steuer-Register

                \-----------/   !
                 andere Chips   --Chip select
                 deaktivieren
```

Wie sind diese Ports des 8255 im CPC nun angeschlossen ?

```
          Bit7     Bit6     Bit5     Bit4     Bit3     Bit2     Bit1     Bit0

Port A      \                                                              /
(R/W)         -------   (Steuerbefehle ->Soundchip, siehe Kap. 2.2) ------


Port B    Tape    Printer  Exp.-    CRTC:    \-- Firmenname  -- /     VSync
(R/-)     Read             Con.   PAL/SECAM  der Einschaltmeldung     CRTC


Port C    Sound    Sound    Tape   Tape-Motor \----   Tastatur Y0-9   ----/
(-/W)     BDIR     BC1      write  0=off;1=on
            !        !
--------    !        !   -----------------------------------------------------
           \!/      \!/


            0        0  =    inaktiv, PSG - Datenbus hochohmig

            0        1  =    Read PSG-Register (Lesen)

            1        0  =    Write PSG-Register (Schreiben)

            1        1  =    Latch PSG-Register -> PSG
```

2.2 Der PSG (AY-3-8912)

Der Soundpufferbereich erstreckt sich von &B700 bis &B7FF. Die Ausgangsfrequenz 62,5 kHz wird heruntergeteilt (Teiler in den Registern R00 bis R05) bis die gewünschte Frequenz entsteht. Daraus ergibt sich :

Periode = 62500 / Tonfrequenz.

Registerbeschreibung :

```
R00 : Kanal A Periodendauer fein           R01 : grob (unbenutzt)
R02 : Kanal B Periodendauer fein           R03 : grob (unbenutzt)
R04 : Kanal C Periodendauer fein           R05 : grob (unbenutzt)

R06 : Rauschen (Bit 0-5)

R07 : Bit 0 : Kanal A \
      Bit 1 : Kanal B  Ausgabe  0=ein;1=aus
      Bit 2 : Kanal C /
      Bit 3 : Kanal A \
      Bit 4 : Kanal B  Rauschen 0=ein;1=aus
      Bit 5 : Kanal C /
      Bit 6 :  Port A 0 = Eingang; 1 = Ausgang --> Port A f. Tastatur-Eingang
      Bit 7 :  Port B 0 = Eingang; 1 = Ausgang

R08 : Kanal A \
R09 : Kanal B   Bit 4 = 0 : normal   -> Bit 0-3 : Lautstärke
R10 : Kanal C / Bit 4 = 1 : ENVelope -> Bit 0-3 : unbenutzt

R11 : ENV Periodendauer Lowbyte
R12 : ENV Periodendauer Highbyte

R13 :   Bit3      Bit2      Bit1      Bit0        Kurve
      Continue   Attack  Alternate   Hold

         0         0         -         -        \___________

         0         1         -         -        /|__________

         1         0         0         0        \|\|\|\|\|\|

         1         0         0         1        \-----------
                                                
         1         0         1         0        \/\/\/\/\/\/
                                                  __________
         1         0         1         1        \|

         1         1         0         0        /|/|/|/|/|/|
                                                 ___________
         1         1         0         1        /

         1         1         1         0        /\/\/\/\/\/\

         1         1         1         1        /|__________


R14 : Tastatur-Eingang X0-9
```

Um nun Daten zum PSG zu transportieren, muss erst das Register gewählt (Reg.-Nr. an Port A des 8255), und eingelatcht werden (Port C des 8255 Bit 6 und Bit 7 auf 1 setzen). Was heisst das? Ein Latch ist prinzipiell eine Speichereinheit, d.h. die Daten wandern zuerst ins Latch (während schon wieder neue Daten anliegen können) und wandern dann schliesslich in den PSG. Wie sieht so etwas aus ?

```
di           ; Interrupts sperren
ld b,&f4     ; Register-Nummer (im Akku)
out (c),a    ; auf Port A des 8255
ld b,&f6     ; Zustand Port C holen
in (a),c     ; (andere Bits sollen unverändert bleiben)
or &c0       ; Bit 6+7 setzen (Latch PSG -> PSG)
out (c),a    ; und schreiben.
and &3f      ; Bit 6+7 zurücksetzen
out (c),a    ; und schreiben. Jetzt ist das PSG-Register ausgewählt.
ld b,&f4     ; Jetzt die Daten (im C-Register)
out (c),c    ; auf Port A des 8255 legen,
ld b,&f6     ; und von Port C
ld c,a       ; wieder die beiden
or &80       ; Bits setzen
out (c),a    ; und schreiben,
out (c),c    ; zurücksetzen und schreiben. Jetzt sind die Daten drin.
ei           ; Interrupts reaktivieren,
ret          ; und Ende.
```

Dies ist original die Firmware-Routine &BD34, wobei im Akku das PSG-Register und im C-Register die Daten übergeben werden. Man kann auch den Firmware- Call &BCAA (SND Release) verwenden, wobei in HL die Adresse einer Tabelle steht, die folgendermassen aussieht :

```
Byte 00    : Kanal    -------> Bit 0 : Kanal A
Byte 01    : ENV-Kurve         Bit 1 : Kanal B
Byte 02    : ENT-Kurve         Bit 2 : Kanal C
Byte 03/04 : Frequenz          Bit 7 : Play direct
Byte 05    : Rauschen
Byte 06    : Lautstärke
Byte 07/08 : Dauer
```

3. Datenträger

3.1. Laden und Speichern

Wie man in BASIC lädt und speichert, ist ja im allgemeinen bekannt. Da stösst man doch öfter auf das "Memory full"-Problem, dazu gibt es einen Trick :
POKE &AC04,&F1 bewirkt, dass nach einer Fehlermeldung nicht abgebrochen wird.
Wie löst man das Problem aber in MC/ASM ?

Laden :

```
                  ld hl,@name         ; (Adresse d. 1. Buchstabens)
                  ld de,<Startadr.>   ; Startadresse in DE
                  ld b,<Anz.>         ; (Anzahl der Buchstaben des Namens)
                  call &BC77          ; (open-load)
                  ex de,hl            ; Startadr. in HL für &BC83
                  call &BC83          ; (load)
                  call &BC7A          ; (close-load)
                   .
                   .hier kann das Prg. mit call <Startadr.> aufgerufen werden
                   .
            @name dm "<name>"         ; hier muss der Name stehen
```

Speichern :

```
                  ld hl,@name         ; \
                  ld de,<Startadr>    ;  wie beim Laden
                  ld b,<Anz.>         ; /
                  call &BC8C          ; (open-write)
                  ex de,hl            ; Startadresse in HL

                  ld de,<Prg.länge>   ; Länge in DE
                  ld bc,<Autostart>   ; Autostart in BC
                  ld a,<filetyp>      ; Filetyp in A
                  call &BC98          ; (write)
                  call &BC8F          ; (close-write)
                      .
                      .
                      .
```

Der Firmware-Call &BC83 liefert als Ausgangswert in HL die Autostart-Adresse, weshalb zum Aufruf des Programms in einem Loader nach dem CALL &BC83 auch ein PUSH HL stehen, und der Firmware-Call &BC7A mit einem JP angesprungen werden kann (siehe hierzu Kapitel 5.3)

3.2. Daten über Files

Wie erhält man Daten über Files ? Man versucht sie zu laden und fragt dann folgende Speicherstellen ab :

	Datasette	**AmsDOS**	**VDOS**
Name	**&B88C-&B89B**	**&A756-&A765**	**&A793-&A7A2**
Start	**&B8A1/&B8A2**	**&A76A/&A76B**	**&A7A7/&A7A8**
Länge	**&B8A4/&B8A5**	**&A76D/&A76E**	**&A7AA/&A7AB**
Autostart	**&B8A6/&B8A7**	**&A76F/&A770**	**&A7AC/&A7AD**
Ladeadr.	**&B89F/&B8A0**	**&A768/&A769**	**&A7A5/&A7A6**
Filetyp	**&B896**	**&A767**	**&A7A4**

Der Versuch zu laden genügt eigentlich schon, man kann aber in MC auch ein kleines Prg. schreiben, das mit **CALL <adr>,@a$** aufgerufen wird, in a$ der Dateiname. Wenn das File existiert, bleibt der String erhalten, wenn nicht wird er gelöscht. Anhand dessen kann entschieden werden, ob die Daten über das File überhaupt ausgelesen werden müssen.

```
        push af       ; \
        push bc       ;  \
        push de       ;  / Register sichern (BASIC ist da zimperlich)
        push hl       ; /
        ld a,&C9      ; Die Bildschirmausgabe
        ld (&BB5A),a  ; blockieren (falls File nicht vorhanden)
        ex de,hl      ; In DE ist @a$, soll nach HL
        push hl       ; HL merken
        ld a,(hl)     ; Ersten Wert holen (Länge des Strings)
        ld b,a        ; und zum Laden ins B-Register
        inc hl        ; nächsten Wert holen
        ld e,(hl)     ; (=Lowbyte der Adresse d. 1. Buchstabens des Strings)
        inc hl        ; nächsten Wert holen
        ld d,(hl)     ; (=Highbyte der Adresse d. 1. Buchstabens des Strings)
        ex de,hl      ; ermittelte Adresse jetzt von DE nach HL
        call &BC77    ; öffnen
        call &BC7A    ; schliessen
        push af       ; Rückmeldungswert merken
        ld a,&CF      ; Erstmal Bildschirmausgabe
        ld (&BB5A),a  ; wieder in Ordnung bringen
        pop af        ; Rückgabewert wieder holen
        pop hl        ; Adresse des Strings vom Stapel nach HL
        cp &FF        ; Rückgabewert = &FF ?
        jr z,@ok      ; JA -> File existiert -> Ende
        ld a,0        ; Nein -> Stringlänge löschen
        ld (hl),a     ; (dadurch wird auch String gelöscht)
   @ok  pop hl        ; \
        pop de        ;  \
        pop bc        ;  / Alte Registerinhalte wiederherstellen
        pop af        ; /
        ret           ; Ende.
```

3.3. Die Datasette

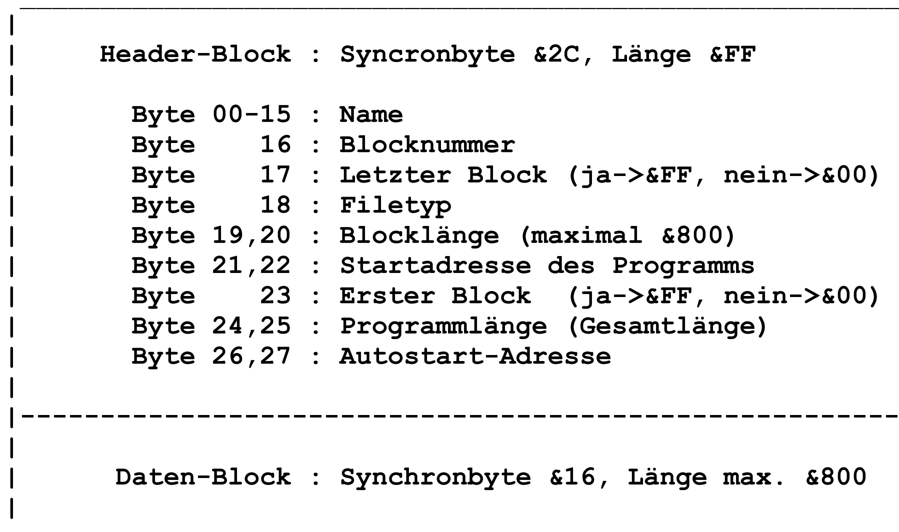
Die Datasette ist ja das in den CPC464 eingebaute Kassetten-Laufwerk, welches es ermöglicht, auf gewöhnliche Musik-Kassetten Daten zu sichern. Die gesamte Ansteuerung läuft indirekt über den Portbaustein 8255.

3.3.1. Blockaufbau

Es existieren die zwei (bekannten) Firmware-Routinen &BCA1 (Block load) und &BC9E (Block save), die den Speicherinhalt an einem Stück schreiben. Dabei wird in HL die Startadresse, in DE die Länge und in A ein Synchronbyte übergeben. Man nennt so einen Block auch "Headerless", weil der gewöhnliche LOAD- und SAVE-Befehl nämlich einen Datenbereich zusätzlich in Blöcke unterteilt, wobei jeder Block wiederum aus einem Header-Block und einem Datenblock besteht. Dies dient der Datensicherheit und hilft auch, ein Programm auf Datasette schneller zu finden, da sich jeder Datenblock ausweist ("FOUND .... block xx"). Der Header eines beliebigen Blocks auf Datasette lässt sich also ganz einfach auslesen :

```
ld hl,<adresse>
ld de,&00FF
ld a,&2C
jp &BCA1
```

Dabei ist der Header wiefolgt aufgebaut :



```
Header-Block : Syncronbyte &2C, Länge &FF

  Byte 00-15 : Name
  Byte    16 : Blocknummer
  Byte    17 : Letzter Block (ja->&FF, nein->&00)
  Byte    18 : Filetyp
  Byte 19,20 : Blocklänge (maximal &800)
  Byte 21,22 : Startadresse des Programms
  Byte    23 : Erster Block  (ja->&FF, nein->&00)
  Byte 24,25 : Programmlänge (Gesamtlänge)
  Byte 26,27 : Autostart-Adresse
--------------------------------------------------
 Daten-Block : Synchronbyte &16, Länge max. &800
```

Die Filetypnummer bedeutet dabei folgendes :

00 : BASIC
01 : BASIC geschützt (mit ,p)   (zum Aufheben siehe Kapitel 4.1 BASIC)
02 : MC
03 : MC   geschützt (??)
22 : ASCII

Dabei wird der Motor immer gestartet und eine Zeit gewartet. Wenn man mehrere Headerless-Blöcke an einem Stück speichern möchte, so wie dies auch bei den gewöhnlichen Datei-Blöcken ist, so muss man mit dem Firmware-Call &BC6E den Motor einschalten, der dann solange eingeschaltet bleibt, bis man ihn mit dem Firmware-Call &BC7D wieder ausschaltet.

3.3.2. Baudrate

Ein Baud sind 1 Bit pro Sekunde. Die Baudrate errechnet sich aus 333333/halbe Länge eines 0-Bits. Bei 1000 Baud ist ein 0-Bit also 666 Microsekunden, ein 1-Bit genau doppelt so lang. Die Elektronik im Rekorder versucht, die Zeitunterschiede zwecks Gleichlauf auszugleichen, daher muss vorkompensiert werden (0-Bits kürzer aufzeichnen, 1-Bits länger). Diese Zeit wird im Akku übergeben, in HL die Baudrate, ein Firmware-Call nach &BC68 stellt diese Baudrate dann ein.

Es ist interessant damit herumzuexperementieren. Dabei ist es empfehlenswert, den Wert im Akku immer grösstmöglich zu halten. Hier ein paar Werte, die u.a. bis an den Bereich des Möglichen gehen :

HL	**&1FB**	**&1C0**	**&150**	**&FA**	**&A5**	**&74**	**&5D**	**&54**	**hl>&230:Baudrate kippt (0=1, 1=0)**
A	**8**	**&10**	**&10**	**&10**	**&10**	**&10**	**&0A**	**&03**	
	lowest	**low**	**Spw.0**	**Cassy**	**Spw.1**	**fast**	**very f.**	**fastest**	

3.3.3. Lese-/Schreibfehler

'Read error a' - zeigt an, dass ein Bit gelesen wurde, dessen Zeit zu lang für die errechnete Zeit für Null- und Eins-Bits ist.

'Read error b' - Jeder Block (auch Headerless) wird intern in Segmente unterteilt mit 256 Bytes Daten + Checksummen-Bytes. Stimmt die gelesene Checksumme nicht mit der errechneten überein, erscheint diese Fehlermeldung.

'Read error d' - Eine eher seltene Fehlermeldung, die anzeigt, dass ein gelesener Block länger als die zulässigen &800 Bytes ist. Dies kann unter Umständen auftreten, wenn der Header-Block manipuliert wurde und dort grössere Werte als erlaubt eingetragen werden.

3.3.4 Kopierschutz-möglichkeiten

Hier kann man sich um das Wissen über den 8255 (Kap. 2.1) nun nicht mehr drücken, denn ohne das kommen wir nicht weiter. Sobald das Band läuft (REC+PLAY gedrückt) nimmt es alle Pegel, gesteuert von Bit 5, auf. Bit 4 bestimmt den Motorzustand, der ja eingeschaltet sein muss, sodass wir auf zwei Grundwerte für Port C des 8255 kommen :

```
0 0 0 1 0 0 0 0 = &10 (TTL-Low-Pegel)
0 0 1 1 0 0 0 0 = &30 (TTL-High-Pegel)

    ! !
    ! !--------- Motor
    !
    !----------- TTL-Pegel
```

Bei den High- und Low-Pegeln handelt es sich nicht etwa um die zwei Frequenzen (Töne) für Lowbits (0) und Highbits (1), sondern um Spannungspegel. Die Frequenz muss man selbst erzeugen, indem man den TTL-Pegel ständig wechselt, aber er darf eben nicht zu schnell oder langsam wechseln, da sonst ungeeignete Frequenzen entstehen. Wechseln wir den Pegel schneller, erhalten wir eine höhere Frequenz, ein langsamerer Wechsel ergibt eine tiefere Frequenz. Meist benutzt man die tiefere Frequenz für das Lowbit, da diese Frequenz leichter wiedererkennbar ist und der 0-Pegel kürzer ist (Siehe Baudrate).

Folgendes kleine Programm erzeugt einen kurzen Ton, der beispielsweise ein Low-Bit sein könnte :

```
         di            ; Interrupts sperren
         ld b,&F6      ; Port C, 8255
         ld c,&70      ; Länge des Tons
         ld a,&10      ; Pegel
 @loop   out (c),a     ; schreiben
         xor &20       ; Pegel wechseln (&10 XOR &20 = &30 und umgekehrt)
         push bc       ; BC merken
         ld b,&A0      ; \
 @delay  djnz @delay   ; / Zeitverzögerung - bestimmt die Tonhöhe
         pop bc        ; BC wieder holen
         dec c         ; C = C - 1
         jr nz,@loop   ; C = 0 ? Nein ? --> Weitermachen
         ei            ; Ja --> Interrupts entsperren
         ret           ; und Ende.
```

Setzt man mei der Zeitverzögerung für B verschiedene Werte ein (z.B. &A0, &C0, &60, &38) so erhält man verschiedene Tonhöhen. Es ist auch wichtig, dass die Interrupts gesperrt werden (Tastaturabfrage, usw.) denn wenn der Computer zwischendrin etwas anderes erledigt, wird die Zeitverzögerung geändert, und es ist tatsächlich hörbar - der aufgezeichnete Ton klingt dreckig und verzerrt und ist zur Datensicherung natürlich völlig ungeeignet. Deswegen ist es auch ein wahres Kunststück (Eine Frage des Timings mal wieder), während des Ladens z.B. eine Musik oder ein Mini-Spiel laufen zu lassen.

Ein witziger Kopierschutz wäre es eigentlich, fünf verschiedene Töne zum Aufzeichnen zu verwenden und die Bytes ins Fünfer-Zahlensystem umzurechnen. Aber bleiben wir bei der Sache. Nachdem wir die Töne nun aufgezeichnet haben, brauchen wir ein erstes Programm, mit dem wir sie "messen" können, um zu unterscheiden, ob es sich um Hi- oder Lo-Bit handelt.

Dieses könnte so aussehen :

```
        di               ; Interrupts sperren
        ld bc,&F610      ; Motor
        out (c),c        ; einschalten
        ld b,&F5         ; Port B des 8255
        in (c),c         ; lesen in C
        res 0,c          ; Bit 0 löschen (VSync-Impuls des CRTC)
@repeat in a,(c)         ; nochmal Port lesen in A
        res 0,a          ; Wieder Bit 0 löschen
        cp c             ; A und C gleich ?
        jr z,@repeat     ; ja -> nix passiert -> kein Pegel -> zurück
        ld e,0           ; In E steht wieoft ein Wert gelesen werden soll,
@loop   dec e            ; bevor er ausgegeben wird (hier &100)
        jr z,@done       ; E auf Null ? Ja - > Wert zeigen
        ld d,0           ; D ist der Zähler mit dem gemessen wird
        in c,(c)         ; Port B lesen (in C)
        res 0,c          ; Vsync raus
@loop2  inc d            ; Zähler erhöhen
        in a,(c)         ; 2. Wert Port B lesen (in A)
        res 0,a          ; Vsync raus
        cp c             ; Werte gleich?
        jr nz,@loop2     ; Nein - > Pegel ändert sich -> weitermessen
        jr @loop         ; Ja - > Pegel ändert sich nicht -> Messung beenden
@done   ld a,d           ; Messwert in A
        set 7,a          ; Kontrollzeichen raus
        and &f0          ; Letzte Ziffer schenken wir uns (variiert zu stark)
        call &bb5a       ; Ausgabe
        jr @loop         ; Neue Messung
```

Nun muss „nur“ noch die Umrechnung von Bits in Bytes und entsprechende Mechanismen zur Korrektur eingefügt werden. Als "Blick über den Tellerrand" - Die vollendete Version eines Turboladers/Savers sieht dann so aus (Achtung - etwas länger) :

```
            jp @lesen          ;          ld h,0
            jp @schreiben      ;          ld b,&b0
@start      dw <startadr.>     ;          jr @wei
@length     dw <länge>         ;          ex af,af'
@sync       db <sync.Byte>     ;          jr nz,@lab1
@place      db 0               ;          jr nc,@lab2
@lesen      di                 ;          ld (ix+0),l
            ld ix,(@start)     ;          jr @ok
            ld de,(@length)    ; @lab1    rlc
            ld a,(@sync)       ;          xor l
            scf                ;          ret nz
```

```
            call @read        ;          ld a,c
            ex af,af'         ;          rra
            xor a             ;          ld c,a
            ld (@place),a     ;          inc de
            pop af            ;          jr @ok2
            ex af,af'         ; @lab2    ld a,(ix+0)
            ei                ;          xor l
            ret c             ;          ret nz
            ld a,1            ; @ok      inc ix
            ld (@place),a     ; @ok2     dec de
            ret               ;          ex af,af'
@schreiben  di                ;          ld b,2
            ex af,af'         ; @wei     ld l,1
            push af           ; @rep3    call @old1
            exx               ;          ret nc
            push bc           ;          ld a,&cb
            ld bc,&F600       ;          cp b
            exx               ;          rl l
            ld ix,(@start)    ;          jp nc,@rep3
            ld de,(@length)   ;          ld a,h
            ld a,(@sync)      ;          xor l
            call @write       ;          ld h,a
            exx               ;          ld a,d
            pop bc            ;          or e
            exx               ;          jr nz,@next
            pop af            ;          ld a,h
            ei                ;          cp &01
            ret               ;          ret
@old1       call @cont1       ; @write   ld hl,&1f80
            ret nc            ;          bit 8,a
@cont1      ld a,&16          ;          jr z,@done
@lp1        dec a             ;          ld hl,&CC98
            jr nz,@lp1        ;          ex af,af'
            and a             ;          inc de
@back1      inc b             ;          dec ix
            ret z             ;          ld a,&30
            ld a,&f5          ;          ld b,a
            in a,(&00)        ; @del2    djnz @del2
            and a             ;          exx
            ret c             ;          out (c),a
            xor c             ;          exx
            and &80           ;          nop
            jr z,@back1       ;          xor &20
            ld a,c            ;          ld b,&a3
            cpl               ;          dec l
            ld c,a            ;          jr nz,@del2
            and 0             ;          ld b,&2E
            and 0             ; @del3    djnz @del3
            and 0             ;          exx
            nop               ;          out (c),a
            scf               ;          exx
            ret               ;          nop
@read       inc d             ;          ld bc,&3A10
            ex af,af'         ;          ex af,af'
            dec d             ;          ld l,a
            ld a,&f5          ;          jp @mx1
            in a,(&00)        ; @next2   ld a,d
            and &80           ;          or e
            ld c,a            ;          jr z,@mx2
            cp a              ;          ld l,(ix+0)
@do         call @cont1       ; @mx0     ld a,h
```

```
            jr nc,@do         ;          xor l
            ld hl,&0415       ; @mx1     ld h,a
 @del1      djnz @del1        ;          ld a,&30
            dec hl            ;          scf
            ld a,h            ;          jp @dbf
            or l              ; @mx2     ld l,h
            jr nz,@del1       ;          jr @mx0
            call @old1        ; @ws      ld a,c
            jr nc,@do         ;          bit 7,b
 @rep       ld b,&9C          ; @del5    djnz @del5
            call @old1        ;          exx
            jr nc,@do         ;          out (c),a
            ld a,&C6          ;          exx
            cp b              ;          nop
            jr nc,@do         ;          ld b,&3d
            inc h             ;          jr nz,@ws
            jr nz,@rep        ;          dec b
 @rep2      ld b,&c9          ;          xor a
            call @cont1       ;          ld a,&30
            jr nc,@do         ; @dbf     rl l
            ld a,b            ;          jp nz,@del5
            cp &d4            ;          dec de
            jr nc,@rep2       ;          inc ix
            call @cont1       ;          ld b,&32
            ret nc            ;          ld a,d
                              ;          inc a
                              ;          jp nz,@next2
              .               ;          ld b,&3b
              .               ; @del6    djnz @del6
         weiter -> 2.Spalte   ;          ret
```

Wer den Border während dem Laden jetzt noch flackern lassen möchte, muss nach dem Befehl IN (C),A folgendes schreiben oder als Unterprogramm anspringen :

```
            push bc           ; BC-Register und
            push af           ; AF-Register sichern
            ld bc,&7f10       ; B=&7F (Gate Array), C=&10 (Border)
            and 8             ; Gelesenes Byte von Datasette
            rlca              ; solange manipulieren,
            rlca              ; bis geeignete Farbwerte
            rlca              ; entstehen und
            res 7,a           ; die oberen Bits
            set 6,a           ; entsprechend
            res 5,a           ; einstellen.
            out (c),c         ; Stift wählen (Border)
            out (c),a         ; Farbwert schreiben
            pop af            ; Alte Register-Werte
            pop bc            ; wiederherstellen.
```

3.4. Die DDI1-Floppy

Die FAT16-Partition des PC ist ja noch gar nicht soo alt - sie ist nämlich interessanterweise ganz ähnlich aufgebaut, wie beim CPC. Der CPC war anscheinend (einer) der erste(n) Homecomputer, der das 8.3-Format führte, (8 Zeichen f. Name + "." + 3 Zeichen Extension) was ja nun (wieder) aus der Mode ist.

Wenn die DDI1-Floppy ausgeschaltet/nicht angeschlossen ist, ist ab &AB7F der Speicher zunächst frei. Ansonsten belegt das Diskettenlaufwerk den Bereich &A700 bis &ABAF und den Bereich &BE40 bis &BEA0. Beim CP/M-Start wird zusätzlich der Bereich &BE80 bis &BEBE mit ROM-Jumps der Floppy gefüllt. Der Speicherbereich &A709 bis &BFFF ist ja software-reset-sicher (ausgenommen der Bereiche die dann neu überschrieben werden (&BDF4-&BB00, &ABFF-&BA6B).

Wenn man sich den Firmware-Bereich für die Ein/Ausgabe näher ansieht, wird man feststellen, dass im Tape-Modus jede Firmware-Routine ihre eigene Adresse hat (siehe z.B. die Vektoren &BC77-&BC9B), die mit einem RST &08 Low Jump angesprungen wird. Im Disc-Modus jedoch verzweigen ALLE diese Firmware-Adressen an &A88B mit einem RST &18 Far Call (--> Sprung nach &CD30 des Floppy-ROMs). Was geht da vor sich? Ab &CD30 im Floppy-ROM steht eine Adress-Verteilerroutine :

```
ld iy,(&BE7D)  ; Adr. d. Floppy-Speichers holen
di             ; Interrupts sperren
ex af,af'      ; \ hier unwesentlich       (Eingangswerte
ld a,c         ; /                          aufarbeiten)
pop de         ; Stapel abbauen
pop bc         ; um an die
pop hl         ; Absprungadresse
ex (sp),hl     ; zu gelangen (-->in HL)
push bc        ; Den Rest wieder
push de        ; auf den Stapel
ld c,a         ; \ hier unwesentlich       (Eingangswerte
ld b,&7F       ; /                           aufarbeiten)
ld de,&10D2    ; Adresse neu berechnen
add hl,de      ; und zurück
push hl        ; auf den Stapel
exx            ; \ hier unwesentlich       (Eingangswerte
ex af,af'      ; /                           aufarbeiten)
ei             ; Interrupts entsperren.
jp &BE7F       ; Patch anspringen (Normalerweise unbenutzt = RET)
                 --> Sprungadresse auf Stapel anspringen.
```

Diese Routine holt sich vom internen Stapel die Absprung-Adresse (z.B. &BC77), und addiert &10D2 dazu (insgesamt &10D5 wg. JUMP), sodass die Adressen &BC77 bis &BC9B zu &CD4C bis &CD70 geändert werden. Dort stehen im Floppy-ROM dann wiederum entsprechende Jumps zu den Zielroutinen.

Ab &A864 im RAM sind die Tape-Vektoren zwischengespeichert, die mit den entsprechenden Befehlen (|TAPE, |DISC, |TAPE.IN, usw...) einfach nach &BC77 bis &BC9B nach Bedarf zurückkopiert werden. Enthalten die Adressen &BC77 und &BC8C also den Wert &CF, ist der Tape-Modus aktiv, währenddessen der Disc-Modus aktiv ist, wenn die den Wert &DF enthalten.

3.4.1. Directory und Disk-Formate

Den Diskettenaufbau darf man sich garnicht mal so kompliziert vorstellen. Das Diskettenformat teilt die Diskette in Tracks (Spuren) und Sektoren ein, wobei ein Sektor in der Regel 512 Bytes umfasst. Es gibt drei Diskettenformate, die hardwaremässig unterstützt werden :

	Format-Nr.	**Sektoren/Track**	**Directory-Track**
IBM Format	**001 (&01)**	**08**	**01**
CP/M / Vendor	**065 (&41)**	**09**	**02**
AmsDOS-DATA	**193 (&C1)**	**09**	**00**

Die Format-Nummer lässt sich der Speicherstelle &A89F (&A8DF für Laufwerk B) entnehmen. Explizit finden wir die Nummer der Directory-Track auch in der Speicherzelle &A89D (&A8DD für LW. B). Nun liest man die ersten vier Sektoren der Directory-Track - - Wie geht das?

Im Floppy-ROM an der Stelle **&C03C** ist ein Jump zu der **Routine "Read Sector".** Diese verlangt als Eingangswerte :

E = Laufwerksnummer (0 = A, 1 = B)
D = Tracknummer (0-42)
C = Sektor - 1 + Format-Nr.
HL = Zieldresse im RAM-Speicher

Wie können wir diese Routine aufrufen? Genau so, wie es bei den Firmware-Calls auch gemacht wird - mit einem RST &18.

```
            rst &18         ; Far Call
            dw @tabelle     ; Tabelle enthält Sprungdaten
            ret             ; Ende
  @tabelle  dw &C03F        ; Zieladresse &C03F
            db &07          ; ROM-Nr. 07 (Floppy)
```

Die Routine &C03F speichert ensprechend einen Sektor.

Schauen wir uns den gelesenen Speicherbereich nun an, erkennen wir, dass jeder Eintrag 32 Bytes umfasst. Die Bedeutung der Bytes ist wiefolgt :

01. Byte	**Usernr. (0 - 255)**
02. - 12. Byte	**Filename mit Extension (Extension in den letzten 3 Bytes)**
13. Byte	**Folgenr.**
17. Byte	**Blocknummern (geteilt durch 2)**

Bei LOAD und SAVE wird dieser Directory-Eintrag nach &A709 kopiert. Wie wir vom C64 wissen, kann man bei der Disketten-Einteilung auch mit Blocks und Records rechnen, das ist weniger physikalisch, jedoch anwenderfreundlicher, da uns die eine Zahl alle Information gibt, die wir brauchen. Wie errechnet man aus der Blocknummer nun Track und Sektor ?

tracknr. = INT (DirectoryTrack + Blocknr. / Sektoren pro Track)

sektornr.= DirectoryTrack * Sektoren pro Track -
Sektoren pro Track * tracknr. + blocknr. + 1

Diese Blocknummern im Directory-Eintrag zeigen dem Computer nach der Reihe, welche Sektoren er auslesen muss, bis das zu ladende Programm komplett im Speicher ist. Grössere Programme oder stark fragmentierte (verstreute) brauchen nun mehr Blocknummern, als in einen Directory-Eintrag passen, daher haben diese Programme mehrere Directory-Eintrage. Hierzu wird die Folgenummer benötigt - sie zeigt, welcher Eintrag nun der erste ist und in welcher Reihenfolge die Einträge abgearbeitet werden müssen. Dabei hat der erste Eintrag die Folgenummer 00, die dann pro Eintrag laufend um 1 erhöht wird.

Liest man nun den ersten Block eines Programms, so findet man weitere Informationen - zuerst nocheinmal die Usernr. und den Namen, dann folgen :

18. Byte — Filetypnr. (siehe Kapitel 3.3)
21. / 22. Byte — Startadresse
24. / 25. Byte — Länge
26. / 27. Byte — Autostart

Bei LOAD und SAVE wird dieser erste Block nach &A756 kopiert. Eine besondere Bedeutung kommt noch dem Extension, also den drei Buchstaben nach dem Punkt zu. Ist zu dem ersten Buchstaben der Wert 128 dazuaddiert, besitzt das File den Status R/O (ReadOnly) (kann nicht gelöscht werden). Trifft diese Addition auf den mittleren Buchstaben zu, ist das Programm unsichtbar (wird im Katalog nicht angezeigt) - (leider gibt es auf dem CPC keinen ATTRIB-Befehl)

Eine Ausnahme bilden wieder die ASCII-Dateien, denen keine Informationen vorangehen, die also gleich ab dem 1. Block beginnen.

Um eine Diskette zu formatieren, kann die Routine &C042 oder &C652 im Floppy-ROM angesprungen werden, welche als Eingangswerte folgende Parameter benötigt :

E = Laufwerksnummer (0 = A, 1 = B)
D = Tracknummer (0-42)
C = Format-Nr.
HL = Adresse im RAM-Speicher

Die Sektoren sind in bestimmter Reihenfolge auf einer Spur gespeichert und haben die Sektor-IDs : 0, 5, 1, 6, 2, 7, 3, 8, 4 (,9 für Fremdformat).

Ab der angegebenen Adresse im RAM-Speicher muss dann eine Tabelle folgen, die folgendermassen aufgebaut ist :

```
trknr  <-- Tracknummer
00     <-- Kopf-Nr. (&00)
x1     <-- Format-Nr. + Sektor-ID 1  (0)
02     <-- Sektorgrösse (2 * 256 Bytes)

trknr  <-- Tracknummer
00     <-- Kopf-Nr. (&00)
x2     <-- Format-Nr. + Sektor-ID 2  (5)
02     <-- Sektorgrösse (2 * 256 Bytes)
.
.
.
```

Um Speicherplatz auf der Diskette zu gewinnen, kann man Spur 40 und 41 ebenfalls formatieren. Dies liefert zusätzlich 9 Kilobytes. Um diese Tracks auch beschreiben zu können, muss zunächst der Disk-Login abgeschaltet werden. Wird eine Diskette geloggt, so werden ihr Format, etc. erkannt und diese Werte in die Speicherbereiche der DDI1-Floppy übertragen. Damit vom Benutzer manipulierte Disketten-Daten nicht vom Login überschrieben werden, der normalerweise bei jedem Disk-Zugriff erfolgt, muss lediglich an die Stelle &A8A8 ein Wert <>0 gesetzt werden (man wählt gewöhnlich &FF). Nun erhöht man den Wert der verfügbaren Kilobytes einer Diskette (dieser Wert steht an der Speicherstelle &A895) um 9 und hat auf einer Diskette mit DATA-Format 187K Speicherplatz. Zum lesen der zwei gewonnenen Tracks ist keine Manipulation von Speicherzellen mehr notwendig und auch der Disk-Login kann wieder aktiviert sein.

Man kann die Diskette aber auch mit einem DATA-Format (&C1) mit 10 Sektoren und 42 Spuren (0 - 41) formatieren und gewinnt 30K, sodass man insgesamt 208K pro Disk-Seite hat. Allerdings wird dieses Fremdformat nicht mehr hardware- mässig unterstützt. Wie kann man solche Disketten nun schreiben und lesen ?

Der Login-Patch (&BE7F) muss auf folgende Routine verbogen werden (mit **POKE &BE7F,&C3:POKE &BE80,<lowbyte Startadr>:POKE &BE81,<hibyte Startadr>**) :

```
            ORG <Startadr.>  ; Startadresse festlegen (Assembler-Steuerbef.)
            di               ;
            ex af,af'        ; L'-Register in A
            exx              ;
            push af          ;
            LD a,l           ;
            ld (@merke1),a   ; absichern
            cp &4c           ; ev.
            jr z,@weiter     ; verwzeigen?
            cp &61           ;
            jr z,@weiter     ;
            cp &70           ; Fremdformat
            jr z,@weiter     ;
            pop af           ; alten Register-
            exx              ; zustand
            ex af,af'        ; herstellen
            ei               ;
            ret              ; Ende
  @weiter   pop af           ;
            exx              ;
            ex af,af'        ;
            ei               ;
            push hl          ;
            push bc          ;
            push de          ;
            push af          ;
            ld a,(&BE7D)     ; Beginn d. Floppy-Speichers ( &A700 )
            rst &18 &C630,7  ;
            ld a,(&BE4C)     ; Kopf-Leseverzögerung
            bit 5,a          ;
            jr z,@wei2       ;
            ld hl,(&BE42)    ; LW-Tab. ( &A890 )
            ld bc,(&BE7D)    ; Floppy-Speicher ( &A700 )
            ld a,(BC)        ;
            cp 0             ; LW A ?
            jr nz,@cont2     ; ja -> Ok.
            ld (@merke1),hl  ; Nein -> einstellen
            jr @wei3         ;
  @cont2    ld de,&0040      ; Adr. + &40
```

```
            add hl,de          ;
            ld (@merke1),hl    ; sichern
@wei3       call @sub          ;
            ld hl,@merke2+9    ;
            ld a,(@merke2)     ;
            cp (hl)            ;
            jr nz,@cont3       ;
            ld hl,(@merke1)    ;
            ld de,&0018        ;
            add hl,de          ;
            ld (hl),0          ;
            jr @ende           ;
@cont3      ld de,(@merke1)    ;
            ld bc,&0019        ;
            ld hl,@puffer      ;
            ldir               ;
@ende       pop af             ;
            pop de             ;
            pop bc             ;
            pop hl             ;
            ret                ;

@sub        ld hl,(@merke1)    ;
            ld de,&16          ;
            add hl,de          ;
            ld d,(hl)          ;
            ld hl,(&BE7D)      ;
            ld e,(hl)          ;
            ld hl,@merke2 - 1;
            push hl            ;
            rst &18 &C763,7    ;   ??
            pop hl             ;
            ld (hl),0          ;
            ld b,&0A           ;
@loop       push hl            ;
            push bc            ;
            rst &18 &C55D,7    ;   ??
            pop bc             ;
            pop hl             ;
            ret nc             ;
            ld a,(&BE51)       ;   Letzter benutzter Sektor in A
            inc hl             ;
            ld (hl),a          ;
            djnz @loop         ;
            ret                ;
@merke2     ds 18              ;
@merke1     ds 2               ;
```

Dadurch, dass die Routine vom Login-Patcch angesprungen wird, wird da Fremdformat nun automatisch erkannt, aber auch die normalen Formate sind weiterhin lesbar. Dies ist sicherlich die bequemste Variante. Leider habe ich zu wenig Informationen, um die Routine ausführlich zu kommentieren. Für Hilfe bin ich dankbar.

3.4.2 Lese/Schreibfehler

Es ist keine grosse Sache - zuerst schaltet man die "Retry, Ignore or Cancel"-Meldung ab indem man die Speicherstelle &BE78 ein Flag setzt (Wert ungleich Null einfügt). Das eventülle "Bad Command", welches noch erscheinen könnte lässt sich dadurch vermeiden, indem man die Firmware-Textausgabe abschaltet mit **POKE &BB5A,&C9**. Nach einem Lesebefehl (z.B. **CAT bzw. CALL &BC9B**) erhalten wir dann in der Speicherstelle &BE4D die Information ueber den Leseerfolg der Diskette :

000 = keine Diskette eingelegt
001 = Unbekanntes Format
002 = Lese/Schreib-Fehler
128 = Alles Ok.

Natürlich sollte man zu gegebener Zeit die Textausgabe durch **POKE &BB5A,&CF** wieder reaktivieren...

3.4.3 Gelöschte Files retten

Löscht man eine Datei, so wird intern einfach die User-Nummer auf &E5 (229) geändert, womit der Platz auf Diskette zunächst wieder freigegeben wird. Solange nichts neues auf die Diskette geschrieben wurde, kann man jederzeit die Usernummer wieder auf 0 setzen und die Datei ist wieder existent. Wie macht man das ? Entweder man macht diese Änderung direkt auf Diskette oder man wechselt in den USER 229. In der Speicherzelle &A701 ist die aktuelle User-Nummer gespeichert. Ich weiss nicht, warum dem Benutzer standardmässig die User 17-255 vorenthalten blieben, jedenfalls kann man mit **POKE &A701,&E5** in den User 229 wechseln, das entsprechende Programm laden, in den User 0 wechseln und es wieder speichern.

Mit der FAT-16 Partition (bis Windows 95) auf dem PC verhält es sich ganz ähnlich. Dort gibt es allerdings keine User mehr, dort wird der erste Buchstabe der Datei mit &E5 überschrieben. Wer sich an DOS-Befehle wie UNDELETE oder an den Norton-Befehl UNERASE erinnert, wird wissen, dass diese immer nach dem ersten Buchstaben gefragt haben, wenn man ein solches Programm retten wollte...

3.4.4 Kopierschutz-möglichkeiten

Hier können natürlich verschiedene Faktoren miteinander verknüpft werden, wie z.B. Fremdformat, CP/M-Start, Verschlüsselung, usw.

Man muss den Teil im Ladeprogramm finden, der das Fremdformat einstellt und das Programm lädt, und vor dem Einsprung zum Start an eine Routine verzweigen, die das alte Format wieder herstellt und den betreffenden Speicherbereich sichert. Ein CP/M-Start kann nur bei Track 0, Sector 1 mit der Startadresse &100 sein, sodass man also schon genau weiss, wo man suchen muss. Die Floppy-ROM-Routinen &C03C und &C03F wurden ja schon erwähnt, sie können auch indirekt mit dem Firmware-Call &BCD4 aufgerufen werden. Man übergibt in A eine Token-Nummer und erhält in HL die gesuchte Adresse. Die Routine '&C03C Read Sector' hat das Token &85, '&C03F Write Sector' hat das Token &84.

Herkömmliche Kopierprogramme (z.B. DISCCOPY unter CP/M) kopieren nur bis Track 39, so würden also bei Disketten mit 42 Spuren zwei Tracks fehlen und das Programm würde nicht mehr einwandfrei laufen, sofern es diesen Platz verwendet hat. Man kann natürlich gezielt gewisse Daten (Loader) direkt auf die Tracks 40 und 41 schreiben (mit &C03C/&C03F), wenn man es nicht dem Zufall (bzw. der Floppy-Routine) überlassen will, wo welche Daten gesichert werden.

In der Speicherzelle &A89D steht ein Flag, das bei CP/M-Disks gesetzt ist. Man kann also auch das CP/M-Betriebssystem einer Diskette überschreiben (mit insgesamt 7K), indem man den Login abschaltet und das Flag setzt (POKE &A8A8,&FF, POKE &A89D,0). Nun hat man eine völlig neue Directory, weil diese bei CP/M-Disks ja erst auf Track 2 sind, nun aber ab Track 0 beginnen, weil wir eine DATA-Diskette vortäuschen. Man hat bei dieser Methode im Endeffekt zwei Directories auf einer Diskette. Ob sich die zweite Directory für 7K lohnt, sei dahingestellt - Fakt ist aber, dass dort Dateien (Loader) sein können, die von der normalen Directory aus absolut unzugänglich sind. Schaltet man den Disketten-Login mit POKE &A8A8,0 wieder ein, so lässt sich die Diskette wie üblich behandeln.

Achtung ! Sollte man das Limit von 7K überschreiten, so werden Dateien der ursprünglichen Directory einfach überschrieben und sind danach nicht mehr lauffähig.

Auch begegnet sind mir schon Fake-Einträge (unechte) in der Directory, die keiner Datei zugeordnet waren und im Datei-Namen Control-Zeichen enthielten, die Schreibstift und Hintergrund gleichsetzten, sodass die Directory ab diesem Punkt zwar weiter aufgelistet wurde, jedoch unlesbar war. Unter Umständen kann man mit dieser Technik sogar einen Screen mit beliebigem kurzen Text anstelle der Directory erscheinen lassen.

Übrigens liess sich ein ähnlicher Trick auch auf dem PC mit der FAT16-Partition vollführen : Man konnte die Kennnummer, die anzeigte ob es sich um eine Datei, um ein Verzeichnis oder um den Datenträger-Name handelte so manipulieren, dass es möglich war, hinter dem Datenträger-Namen ein "geheimes" Verzeichnis zu verstecken. Allerdings wurde dies von SCANDISK bemerkt, sodass man SCANDISK umenennen musste, dafür eine BATCH-Datei mit dem Namen SCANDISK.BAT schreiben musste, die das Verzeichnis vor dem Aufruf von SCANDISK sichtbar machte, und danach wieder unsichtbar, und diese BATCH-Datei eventuell sogar in ein EXE-File umwandelte, damit die Tarnung perfekt war.

Darüberhinaus ist es natürlich, wie bei jedem Hardware-Gerät möglich, vollkommen auf Firmware-Routinen zu verzichten und direkt mit Port-Befehlen (OUTs) zu arbeiten. Einige Disketten-Loader tun das auch tatsächlich. Auch hier bin ich für Hinweise sehr verbunden.

4. Intern

4.1. BASIC

Die Programmiersprache BASIC dient dem CPC ja nicht nur zur Programmierung, sondern auch als Betriebssystem. Das Basic-Betriebssystem beginnt ab &C000 im ROM. Dabei fängt die BASIC-Initialisierung ab &C006 an, der READY-Modus ab &C064. Das erste, was der READY-Modus tut ist, die Speicherstelle &AC01 im RAM mit einem CALL aufzurufen. Dies ist der sogenannte READY-Patch. Ein Patch besteht im allgemeinen aus einem JUMP oder 3 Bytes, die mit RET (&C9) gefüllt sind, so wie dieser READY-Patch. Man kann hier an eine beliebige Routine verzweigen, die nach jeder Eingabe im READY-Modus ausgeführt wird. Nicht zu vergessen ist, dass hier noch das BASIC-ROM eingeschaltet ist, also Sprünge nach &C000-&FFFF nicht auf dem Bildschirm, sondern im BASIC-ROM landen. Programme, die ab &AC01 beginnen, starten daher sofort, auch wenn sie nur mit LOAD geladen werden.

Was nützt uns nun der READY-Patch?
Nach dem Aufruf des Ready-Patches folgen dann noch einige Kleinigkeiten, wie z.B. Sound sperren, Bildschirm initialisieren, geschützte Basic-Programme löschen, usw. bis schliesslich ab &C090 das erste READY auf dem Bildschirm ausgegeben und auf eine Eingabe gewartet wird. 'Verbiegt' man also den Patch nach &C090 ins BASIC-ROM (mit POKE &AC01,&C3:POKE &AC02,&90:POKE &AC03,&C0), so überspringt man all diese Kleinigkeiten, darunter das Löschen von geschützten BASIC-Programmen, womit diese nun listbar werden. Um das BASIC-Programm ungeschützt abzuspeichern, muss nun noch das Flag an der Speicherstelle &AE45 zurückgesetzt werden (POKE &AE45,0), danach kann der READY-Patch auch wieder zurückgesetzt werden (POKE &AC01,&C9).

Darüberhinaus kann man mit dem READY-Patch, bzw. mit dem Error-Patch (&AC04) auch Befehlserweiterungen ohne RSX realisieren. Über den Error-Patch kann man solche Fehler abfangen. Dabei zeigt HL auf eine Stelle weiter des letzten Buchstabens, des fehlerhaften Wortes oder Arguments. Im E-Register ist die Fehlernummer enthalten.

Für BASIC-Programme ist der Speicherbereich &0170 bis &AB80 vorgesehen. Dabei werden die BASIC-Befehle in Tokens zerlegt. Die Zeile '1 CLS' sieht also im Speicher so aus :

```
&0170 &01 \ Zeilennummer
&0171 &00 /
&0172 &06 \ Länge in Bytes
&0173 &00 /
&0174 &8A   Token für CLS
&0175 &00   Jede Zeile wird mit &00 abgeschlossen
```

Welches Token gehört nun zu welchem Befehl ?

Tabelle 1 :

&00 Zeilenende	**&10 Konstante 2**	**&1B 2-Byte Wert bin.**
&01 Trennzeichen (:)	**&11 Konstante 3**	**&1C 2-Byte Wert hex.**
&02 Integer (%)	**&12 Konstante 4**	**&1D Zeilenadresse**
&03 String ($)	**&13 Konstante 5**	**&1E Zeilennummer**
&04 Real (!)	**&14 Konstante 6**	**&1F Fliesskommawert**
&0B \ Variable	**&15 Konstante 7**	
&0C ohne	**&16 Konstante 8**	
&0D / Kennzeichen	**&17 Konstante 9**	
&0E Konstante 0	**&19 1-Byte Wert**	
&0F Konstante 1	**&1A 2-Byte Wert dez.**	

&80 AFTER	**&A0 GOTO**	**&C0 '**	**&E0 (FRAME)**
&81 AUTO	**&A1 IF**	**&C1 RAD**	**&E1 (CURSOR)**
&82 BORDER	**&A2 INK**	**&C2 RANDOMIZE**	**&E2 (CLEAR INPUT)**
&83 CALL	**&A3 INPUT**	**&C3 READ**	**&E3 ERL**
&84 CAT	**&A4 KEY**	**&C4 RELEASE**	**&E4 FN**
&85 CHAIN	**&A5 LET**	**&C5 REM**	**&E5 SPC**
&86 CLEAR	**&A6 LINE**	**&C6 RENUM**	**&E6 STEP**
&87 CLG	**&A7 LIST**	**&C7 RESTORE**	**&E7 SWAP**
&88 CLOSEIN	**&A8 LOAD**	**&C8 RESUME**	**&E8 ---**
&89 CLOSEOUT	**&A9 LOCATE**	**&C9 RETURN**	**&E9 ---**
&8A CLS	**&AA MEMORY**	**&CA RUN**	**&EA TAB**
&8B CONT	**&AB MERGE**	**&CB SAVE**	**&EB THEN**
&8C DATA	**&AC MID$**	**&CC SOUND**	**&EC TO**
&8D DEF	**&AD MODE**	**&CD SPEED**	**&ED USING**
&8E DEFINT	**&AE MOVE**	**&CE STOP**	**&EE >**
&8F DEFREAL	**&AF MOVER**	**&CF SYMBOL**	**&EF =**
&90 DEFSTR	**&B0 NEXT**	**&D0 TAG**	**&F0 >=**
&91 DEG	**&B1 NEW**	**&D1 TAGOFF**	**&F1 <**
&92 DELETE	**&B2 ON**	**&D2 TRON**	**&F2 <>**
&93 DIM	**&B3 ON BREAK**	**&D3 TROFF**	**&F3 <=**
&94 DRAW	**&B4 ON ERROR GOTO**	**&D4 WAIT**	**&F4 +**
&95 DRAWR	**&B5 ON SQ**	**&D5 WEND**	**&F5 -**
&96 EDIT	**&B6 OPENIN**	**&D6 WHILE**	**&F6 ***
&97 ELSE	**&B7 OPENOUT**	**&D7 WIDTH**	**&F7 /**
&98 END	**&B8 ORIGIN**	**&D8 WINDOW**	**&F8 ^**
&99 ENT	**&B9 OUT**	**&D9 ZONE**	**&F9 **
&9A ENV	**&BA PAPER**	**&DA WRITE**	**&FA AND**
&9B ERASE	**&BB PEN**	**&DB DI**	**&FB MOD**
&9C ERROR	**&BC PLOT**	**&DC EI**	**&FC OR**
&9D EVERY	**&BD PLOTR**	**&DD (FILL)**	**&FD XOR**
&9E FOR	**&BE POKE**	**&DE (GRAPHICS)**	**&FE NOT**
&9F GOSUB	**&BF PRINT**	**&DF (MASK)**	**&FF ---> (2. TABELLE)**

Tabelle 2 (Funktionen) :

&71 BIN$

&00 ABS	**&10 LOG10**	**&40 EOF**	**&72 DEC$(**
&01 ASC	**&11 LOWER$**	**&41 ERR**	**&73 HEX$**
&02 ATN	**&12 PEEK**	**&42 HIMEM**	**&74 INSTR**
&03 CHR$	**&13 REMAIN**	**&43 INKEY$**	**&75 LEFT$**
&04 CINT	**&14 SGN**	**&44 PI**	**&76 MAX**
&05 COS	**&15 SIN**	**&45 RND**	**&77 MIN**
&06 CREAL	**&16 SPACE$**	**&46 TIME**	**&78 POS**
&07 EXP	**&17 SQ**	**&47 XPOS**	**&79 RIGHT$**
&08 FIX	**&18 SQR**	**&48 YPOS**	**&7A ROUND**
&09 FRE	**&19 STR$**		**&7B STRING$**
&0A INKEY	**&1A TAN**		**&7C TEST**
&0B INP	**&1B UNT**		**&7D TESTR**
&0C INT	**&1C UPPER$**		**&7E (COPYCHR$)**
&0D JOY	**&1D VAL**		**&7F VPOS**
&0E LEN	**&1E ---**		
&0F LOG	**&1F ---**		

Anmerkungen : - *Die Befehle in Klammern existieren nur auf dem CPC664/6128 und liefern 'Syntax error' oder 'Improper Argument'.*

- *Die Funktion DEC$ ist NICHT im Handbuch erwähnt. Wie aus der Tabelle ersehen werden kann, wurde der Funktion aus Versehen eine Klammer hinzugefügt. Dafür hat sich die Firma anscheinend so geschämt, dass sie sie nicht erwähnt hat. Diese Funktion wandelt eine Zahl in einen String um, wird in der Praxis also beispielsweise so angewendet : a$=dec$((a,"##"). Nicht die zwei offenen Klammern vergessen, eine wird ja noch vom Interpreter hinzugefügt.*

Was man mit diesen Informationen alles anfangen kann, soll an der Aufgabe des Schutzes von BASIC-Programmen gezeigt werden.

1. Möglichkeit :
Man benutzt ein Token, für das kein Befehl existiert. Damit das Programm lauffähig bleibt, muss dieses Token nach einem REM bzw. ' stehen. Dies lässt sich ganz einfach realisieren, indem man sich z.B. mit PRINT CHR$(&E8) ein entsprechendes ASCII-Zeichen ausgeben lässt und dieses mit dem COPY-Cursor zur Befehlszeile nach ein REM kopiert. Ein LIST bewirkt nun ein 'Syntax error' an der entsprechenden Stelle. Wird das REM vergessen, endet der Computer beim Aufruf mit RUN aus lauter Verzweiflung in einer Endlos-Schleife.

2. Möglichkeit :
Man täuscht ein Zeilenende vor. Dies soll nun einerseits der LIST-Routine vormachen, die Zeile sei zu Ende, andererseits sollen die unterschlagenen Befehle jedoch ausgeführt werden. Deshalb geht man wiefolgt vor :

```
Beispiel : 10 CLS:::  GOSUB 5000
                ^
                |--------------- Hier ändert man nun &01 in &00
                                 Die zwei Leerzeichen nach den drei
                                 Doppelpunkten sind wichtig!
```

3. Möglichkeit :
Man macht die erste Zeile unsichtbar, indem man ihr die Zeilennummer 0 gibt, welche nicht gelistet, aber ausgefürt wird. Wie ? Mit POKE 370,0 Ein RENUM macht dies allerdings rückgängig.

4. Möglichkeit :
Man macht eine komplette Zeile unsichtbar, indem man ihre Länge zu der Länge der vorgehenden Zeile addiert, sodass die vorhergehende Zeile diese Zeile komplett überdeckt. Nachteil : Diese Zeile darf nicht mehr angesprungen werden (Zeilennummer existiert nicht mehr).

```
Beispiel :     &0170   01 \ Zeilennr. 1
               &0171   00 /
               &0172   0C \ Länge 1 + Länge 2 (&0C = eigene Länge + &06)
               &0173   00 /
               &0174   8A   Befehl(e) 1
               &0175   00   Ende
               &0176   02 \ Zeilennr. 2
               &0177   00 /
               &0178   06 \ Länge 2
               &0179   00 /
               &017A   8A   Befehl(e) 2
               &017B   00   Ende
```

5. Möglichkeit :
Ab &0040 stehen die Tokens, der Eingabezeile im Direktmodus. Beim Befehl **RUN"<...>"** steht an der Stelle &0041 der Wert &22 für das Anführungszeichen nach dem RUN. Wenn man nun möchte, dass das Programm nur direkt mit **RUN"<...>"** startbar ist und nicht mit 'LOAD"<...>"' und 'RUN', so fragt man diese Speicherzelle in BASIC ab und verzweigt bei nicht vorhandenem Wert &22 an ein END, einen CALL 0 oder ähnliches.

Gerne wird auch die ESC-Taste gesperrt mit **KEY DEF 66,0,0,0** und **POKE &BDEE,&C9**, oder es wird ein **CALL 0** auf den READY-Patch gelegt (mit **POKE &AC01,&C3:POKE &AC02,0:POKE &AC03,0**), oder es wird der LIST-Patch gekillt (mit **POKE &AC13,&F1**)

6. Möglichkeit :
Eine ganz elegante Methode ist es ja, das komplette BASIC-Programm an einen anderen Speicherbereich zu kopieren und als MC abzuspeichern. Natürlich muss davor erstmal ein kleines (echtes) MC sein, dass diese Tokens wieder nach &170 verschiebt (mit LDDR) und dem BASIC-Pointer in &AE83 die entsprechende Endadresse des BASIC-Programms zurückgibt. Anschliessend muss das BASIC Programm ja nun auch gestartet werden mit einem RST &18 nach &E9BD ins BASIC-ROM (Romnummer &FD) und das ganze MC muss natürlich auch mit RUN startbar sein indem man an seinen Anfang folgendes schreibt :

```
LD HL,&AB7B
LD DE,&40
CALL &BCCB
```

Viel praktischer ist allerdings umgekehrt das Einbinden von MCs in BASIC-Programme. Dies bietet sich für Loader, komprimierte Bilder, Demos, usw. an. Wie macht man das? Man schaut mit **PRINT PEEK(&AE83)+256*PEEK(&AE84)**, wo das BASIC-Programm zu Ende ist addiert dann so ca. 10 Bytes hinzu, rundet das ganze auf eine glatte Adresse (&1C0 oder &200 oder &500 oder so) und lädt sein MC nun an diese Adresse (Bei 'Memory full' den Error-Patch ändern, siehe Kapitel 3.1). Damit dieses implantierte MC nun beim Speichern des BASIC-Programms MIT-abgespeichert wird, muss in &AE83 / &AE84 die neue Endadresse (also Adresse des letzten Bytes des MCs + 1) geschrieben werden. Fertig. Auf diese Weise sind MCs auch viel einfacher zu kopieren, man benötigt keine Daten (Startadr., Länge, usw.) mehr. Ausserdem kann man so auch MCs geschützt abspeichern. Es existert zwar der Filetyp 03 (MC protected), ich habe aber nie feststellen können, das dieses MC vor dem Einsprung in den READY-Modus gelöscht wird.

4.2 RSX-Befehle

Durch RSX (Resident System eXtension) wird es dem Benutzer ja ermöglich, Routinen einen Aufrufnamen für BASIC zu geben, was ja komfortabler wie irgendein Call ist, von dem man die Adresse wissen muss und der im Zweifels- fall sogar nicht im zentralen Speicherbereich liegt. Grundsätzlich funktioniert die Initialisierung einer RSX folgendermassen :

```
        ld hl,@platz         ; HL auf reservierte 4 Bytes
        ld bc,@table         ; BC auf Tabelle
        jp &BCD1             ; RSX initialisieren + Ende
                             ; Tabelle muss enthalten :
 @table dw @namen            ; Adresse wo die Namen stehen
        jp <adr1>            ; Jump zu Routine 1
        Jp <adr2>            ; Jump zu Routine 2
        jp <adr3>            ;       .
            .                ;       .
            .                ;       .
            .                ;
 @namen dm "<name1>"         ; Hier die Namen der Befehle,
        db <ASCII-Wert>+128  ; Bei letztem Buchstaben Bit 7 setzen
        dm "<name2>"         ; um Ende zu markieren.
        db <ASCII-Wert>+128  ;       .
            .                ;       .
            .                ;       .
            .                ;
        db 0                 ; Am Ende all der Namen ein 0-Byte
 @platz dw &0000             ; Hier die 4 reservierten
        dw &0000             ; Bytes, die intern verwendet werden.
                             ;
 <adr1>     .                ; Hier können die eigentlichen
            .                ; Routinen beginnen
            .                ;       .
 <adr2>     .                        .
            .                        .
```

Für die Parameterübergabe durch

Call <adr>,<wert-1>,<wert-2>,<wert-3>....,<wert-n> oder
|<BEFEHL>,<wert-1>,<wert-2>,<wert-3>....,<wert-n>

gilt : Jeder Wert, egal wie klein, wird in 16 Bit abgelegt. Das "Pferd" wird dabei "von hinten aufgezäumt" :

Es steht der erstletzte Wert in (IX+0),(IX+1), (letzter Wert auch immer in DE)
der zweitletzte Wert in (IX+2),(IX+3),
der drittletzte Wert in (IX+4),(IX+5),
.
.
.

So kann man den zweitletzten Wert z.B. mit LD l,(ix+2) und LD h,(ix+3) ins HL-Register geladen werden. Im Akku wird die Anzahl der übergebenen Werte abgelegt, sodass auch ein Vergleich möglich ist, ob genug Werte übergeben wurden, um die entsprechende Routine auszuführen.

Strings übergibt man bekannterweise mit @<stringname> also z.B. |era,@a$. Dies ist eine Adresse, an der im 1. Byte die Länge des Strings und dann im 2. und 3. Byte eine weitere Adresse steht, die auf den ersten Buchstaben des Strings zeigt. (Ein kleines Beispiel hierzu ist im Kapitel 3.2 zu finden).

4.3. Interrupts

Interrupts (Unterbrechungen) ermöglichen es, mehrere Dinge "gleichzeitig" geschehen zu lassen, (z.B. Musik spielen, Laufschrift usw). Natürlich werden sie eigentlich nacheinander abgearbeitet, nur geschieht dies so schnell, dass es dem Benutzer "gleichzeitig" erscheint. Die momentane Hauptarbeit der CPU wird also UNTERBROCHEN (daher Interrupt) und es wird eine Routine bearbeitet, danach wird die Haupt-Arbeit fortgesetzt. Dies heisst also, der eigentliche Programmverlauf, sei dies der READY-Modus oder das Abarbeiten eines Maschinen-programms wird dadurch nicht gestört, Registerwerte werden jedoch durch ein Interrupt nicht automatisch gesichert.

Durch einen Interrupt wird also Rechenleistung durch ein Taktsignal vom Prozessor abgezweigt, die beliebig verwendet werden kann. Natürlich müssen die entsprechenden Routinen, die an einem Interrupt hängen auch so geschrieben sein, dass sie PRO AUFRUF etwas um einen Schritt vorantreiben (z.B. Musik : den nächsten Ton spielen odeer Laufschrift : die nächste Pixel reinscrollen). Solche Routinen, die an Interrupts hängen nennt man 'Events'.

Es gibt prinzipiell drei verschiedene Interrupt-Arten : Framefly (gesteuert vom Strahlenrücklauf, alle 1/50 s), Fast Ticker (Vom Timer, alle 1/300 s) und Ticker (Vom Fast Ticker runtergeteilt, alle 1/50 s). Wenn die Standard- Interrupts der Hardware (Tastaturabfrage, Bildschirmaufbau,...) alle gelaufen sind, hangelt sich das System weiter zu den User-Interrupts (falls vorhanden). An der Stelle &B18C (Framefly, bzw. &B18E Fastticker o. &B190 Ticker) im Speicher ist der jeweilige erste Ursprungs-Eventblock dieser Interrupts.

Zu einem CPC-Interrupt-Event gehört IMMER ein Eventblock, der wiefolgt aussieht :

```
                 Byte 00 + 01  reserviert für Ticker-Liste
 nur vorh.     / Byte 02 + 03  reserviert für Ticker-Counter
bei KL Ticker \ Byte 04 + 05  reserviert für Ticker-Reload
                 Byte 06 + 07  reserviert für Pending-Queue
                 Byte 08       Zähler
                 Byte 09       Klasse
                 Byte 10 + 11  Startadresse der Routine
                 Byte 12       ROM-Selection
```

Dabei ist Byte 09 (Klasse) wiefolgt aufgeschlüsselt :

```
Bit 0   : 0 = near (RAM), 1 = ROM
Bit 1-4 : Priorität
Bit 5   : muss immer 0 sein
Bit 6   : 1 = Express (höchste Priorität), 0 = normal
Bit 7   : 1 = asyncron, 0 = syncron
```

Durch Byte 00 + 01 jedes Eventblocks sind alle Interrupts miteinander verkettet, dort steht die Adresse des nächsten Eventblocks. Die "Pending-Queue" ist eine ebensolche lineare Liste aller AKTIVEN Interrupts. Man kann die Event-Blocks nun in die Pending-Queue einfügen oder rausnehmen, um sie zu aktivieren, bzw. zu deaktivieren. Diesen Vorgang bezeichnet man als 'kicken'. Es gibt auch asyncrone Events, sie haben keine Warteschlange, müssen jedoch im zentralen RAM liegen. Der Event-Block kann, muss aber nicht selbst geschrieben werden.

Die **Firmware-Routine &BCD7 (KL new framefly)** erstellt einen solchen Eventblock für eine beliebige Routine und aktiviert sie auch gleich. Dabei benötigt diese Firmware-Routine folgende Eingangswerte :

```
hl : Adresse, wo der Eventblock angelegt werden soll
de : Startadresse der Routine
 c : ROM-Nummer (&FF = RAM)
 b : Klasse (z.B. &80 für asyncrone Interrupt)
```

Zum Aushängen durch die Firmware-Routine KL DEL FRAMEFLY werden nur die Werte im hl- und de-Register benötigt. (Ein Beispiel zu Interrupts in Kapitel 1.5)

4.4. Daten über den Computer

Computertyp feststellen : Auslesen der Speicherstelle &BBFA.

```
(&BBFA) = &36 CPC 464
(&BBFA) = &A2 CPC 664
(&BBFA) = &A6 CPC 6128
```

ROM-Nummer feststellen :

```
LD c,0
CALL &B915
LD a,h         (ROM-Nummer im Akku)
```

4.5. Control-Zeichen

NAME	CTRL-	CHR$(x)	Funktion
NUL		00	Kein Einfluss
SOH	A	01	Symbol an den Drucker schicken
STX	B	02	Textcursor (bei Eingaben) ausschalten
ETX	C	03	Textcursor (bei Eingaben) einschalten
EOT	D	04	Mode (0,1,2)
ENQ	E	05	Nachfolgendes Zeichen an Grafik-Pos. ausgeben
ACK	F	06	Bildschirmausgabe einschalten
BEL	G	07	Piepton (Bell)
BS	H	08	Cursor nach links
TAB	I	09	Cursor nach rechts
LF	J	10	Cursor nach unten (Line Feed)
VT	K	11	Cursor nach oben
FF	L	12	Bildschirm löschen
CR	M	13	Cursor an den Anfang der Zeile (Carriage Return)
SO	N	14	Hintergrundfarbe (Paper) einstellen
SI	O	15	Vordergrundfarbe (Pen) einstellen
DLE	P	16	Zeichen auf Cursorposition löschen
DC1	Q	17	Löschen von Zeilenanfang bis Cursurposition
DC2	R	18	Löschen von Cursorposition bis Zeilenende
DC3	S	19	Löschen von Zeile 1 bis Cursorposition
DC4	T	20	Löschen von Cursurposition bis letzte Zeile
NAK	U	21	Bildschirmausgabe ausschalten
SYN	V	22	Transparentmodus ein/aus
ETB	W	23	Grafikmodus wählen (0=normal, 1=AND, 2=XOR)
CAN	X	24	Tauschen von Hinter- und Vordergrundfarbe
EM	Y	25	gleicht dem BASIC-Befehl SYMBOL
SUB	Z	26	gleicht dem BASIC-Befehl WINDOW
ESC	[	27	kein Einfluss
FS	\	28	gleicht dem BASIC-Befehl INK
GS	]	29	gleicht dem BASIC-Befehl BORDER
RS	/	30	Cursor in linke, obere Ecke setzen
US	0	31	gleicht dem BASIC-Befehl LOCATE

Zur Parameterübergabe :

Die Parameter können entweder als direktes Zeichen übergeben werden, sodass der Parameterwert also gleich dem ASCII-Wert ist, oder es kann manchmal auch direkt die Zahl dahinter geschrieben werden (wg. MOD-Funktion) Es funktioniert also **PRINT CHR$(4)+CHR$(1)** genauso wie **PRINT CHR$(4)+"1"**. Bei mehreren Parametern müssen immer alle Parameter übergeben werden
(z.B. **?chr$(29)+chr$(0)+chr$(0)** für BORDER 0,0).

So wie also beim C64 das berühmte Herzchen zum Löschen des Bildschirms verwendet wird, kann auch hier in einer Printanweisung einiges durch Control-Zeichen angestellt werden. In der Praxis schreibt man nämlich nicht **PRINT CHR$(x)+CHR$(y)+...** sondern **PRINT"<zeichen1><zeichen2>...Text....",** wobei <zeichen1> und <zeichen2> durch gleichzeitigen Druck auf die Control-Taste und den entsprechenden Buchstaben erzeugt werden.

4.6 Einfache Verschlüsselungstechnicken

Die Verschlüsselung (Codierung) eines Proogramms dient in den wenigste Fällen zum Packen (wurde aber in seltenen Fällen auch auf dem CPC schon für Programm Files angewendet), sondern primär als Kopier- und POKE-Schutz. Hierzu nutzt man meistens die XOR-Verknüpfung aus. Obwohl es noch andere Möglichkeiten gibt (z.B. Bit-Rotation), beschränken wir uns hier auf die XOR-Verknüpfung (sie auch Kapitel 5.1) Wie funktioniert diese nochmal ?

```
           Byte1 :  0 0 0 1 0 0 0 1  =  &11 = 017
           Byte2 :  1 0 1 1 0 1 0 0  =  &B4 = 180
           -------------------------------------
 Byte1 XOR Byte2 :  1 0 1 0 0 1 0 1  =  &A5 = 165
```

Die Regel dazu lautet : **Wenn NUR Byte1 ODER NUR Byte2 gleich 1 ist, wird auch das Ergebnis 1, ansonsten 0**

Was passiert nun, wenn wir nun das Ergebis mit der gleichen Zahl wieder XOR-verknüpfen ?

```
           Byte1 :  1 0 1 0 0 1 0 1  =  &A5 = 165
           Byte2 :  1 0 1 1 0 1 0 0  =  &B4 = 180
           -------------------------------------
 Byte1 XOR Byte2 :  0 0 0 1 0 0 0 1  =  &11 = 017
```

Wir erhalten wieder den ursprünglichen Wert des ersten Bytes.

Somit kann also Byte2 als Schlüssel und Byte1 als Verschlüsslungsdaten betrachtet werden, was uns zum ersten, ganz einfachen Verschlüsslungsprogramm führt :

```
         ld hl,<startadresse> ; Startadr. des zu codierenden Bereichs
         ld de,<länge>        ; Länge des zu codierenden Bereichs
         ld b,<Schlüssel>     ; Schlüsselbyte für die XOR-Verknüpfung
 @loop   ld a,(hl)            ; Wert holen,
         xor b                ; verknüpfen
         ld (hl),a            ; zurückschreiben
         dec de               ; Länge := Länge -1
         ld a,e               ; \
         or e                 ; / Länge = 0 ?
         ret z                ; Ja   -> Ende
         jr @loop             ; nein -> Nächstes Byte.
```

Eine weiterentwickelte Methode ähnelt einem Reissverschluss. Man verknüpft das erste Byte mit dem Schlüssel, das zweite Byte mit dem ersten, das dritte mit dem zweiten, usw. Wenn man nun das letzte Byte des bereits verschlüsselten Bereichs mit dem Schlüssel verknüpft, erhält man genau DAS Byte, mit dem man beim Entschlüsseln von hinten wieder anfangen muss. Man kann aber auch zuerst das erste mit dem zweiten, das zweite mit dem dritten usw., und zum Schluss das letzte Byte mit dem Schlüssel verknüpfen, und kommt auf folgendes :

```
        jp @encode          ; Aufruf zur Verschlüsselung
        jp @decode          ; Aufruf zur Entschlüsselung
 @encode ld hl,<Startadr>   ; in HL Startadresse
        ld de,<Länge>       ; in DE L nge
        ld a,<Schlüssel>    ; In A  Schlüssel
 @loop1  inc hl             ; das nächste
        ld a,(hl)           ; mit dem
        dec hl              ; vorhergehenden
        xor (hl)            ; verknüpfen
        ld (hl),a           ;
        inc hl              ; nächste Position
        dec de              ; Solange
        ld a,d              ; Länge <>0
        or e                ; weitermachen
        jr nz,@loop1        ;
        pop af              ; Nun den Schlüssel
        xor (hl)            ; mit dem letzten Byte
        ld (hl),a           ; verknüpfen und
        ret                 ; Ende
 @decode ld hl,<Startadr>   ; in HL Startadresse
        ld de,<Länge>       ; in DE Länge
        ld a,<Schlüssel>    ; in A  Schlüssel
        add hl,de           ; Endadresse berechnen (von hinten anfangen)
        xor (hl)            ; Schlüssel mit letztem Byte
        ld (hl),a           ; verknüpfen
        dec hl              ;
 @loop2  inc hl             ; und jetzt vom
        ld a,(hl)           ; Ende bis zum Anfang
        dec hl              ; alle Bytes
        xor (hl)            ; nach obigem
        ld (hl),a           ; Algorithmus
        dec hl              ; verknüpfen
        dec de              ; solange
        ld a,d              ; Länge in DE
        or e                ; ungleich Null
        jr nz,@loop2        ;
        (ret) .
              .
              .
```

Wenn nach der Dekodier-Routine direkt ein Programm folgt, das nach der Entschlüsselung gleich ausgeführt werden soll, und dies ist in der Regel der Fall, so gibt es nur eine Möglichkeit, diesen Code überhaupt zu entschlüsseln, indem man die Entschlüsslungsroutine an eine andere Adresse verschiebt, natürlich den RET-Befehl am Ende einfügt und sie ausführt.

Dies bringt einen Geheimnis-Krämer natürlich auf die Idee, zusätzlich zu der einen XOR-Verknüpfung die Bytes noch mit dem Adresszähler HL zu verknüpfen (jetzt ist der zu (de)codierende Block fixiert) und das Programm mehrere Male rekursiv aufzurufen (nun ist auch die Dekodierroutine fixiert, es wird also falsch entschlüsselt, falls sie an anderer Adresse steht).

Man stelle sich also vor, der Source-Code, der da gerade entschlüsselt wurde und nun abgearbeitet wird, fügt an DIE Stellen, wo Startadresse, Länge und Schlüsselbyte geladen werden (am Anfang) einfach explizit neue Werte ein
(LD HL,<adr> : LD (nn),HL) und startet die Entschlüsslungs-Routine erneut.

Ich glaube, sowas nennt man "Dynamische Programmstruktur", und da sind wir bei den gängigen Verschlüsselungs-Methoden, die gerne in jeder Form von Programm Ladern verwendet werden und die nicht ohne weiteres zu knacken sind, besonders wenn sie sich in Speicherbereichen befinden, die im READY-Modus nicht mehr bearbeitet werden können, weil sie dann überschrieben werden (z.B. &0040) oder weil die Firmware nicht mehr funktioniert (z.B. &BB00)

Man müsste nun zum Dekodieren die Verschlüsselungsroutine woanders hin-kopieren, an die Startadresse der ursprünglichen Verschlüsslungsroutine einen Jump zur kopierten machen und die Werte (Startadr., Länge, Schlüssel) auslesen, die vom entschlüsselten Teil neu eingefügt werden, abgesehen vom Relokatieren...

Wenn dann einer nun noch die Gemeinheit besitzt, in seiner Verschlüsselung mit dem SP-Register (bzw. dem PC-Register) zu verknüpfen, kommt man erst mal ganz schön ins Grübeln...

(- Hilfreiche Hinweise hierzu werden gerne entgegengenommen :)) -)

4.7 CP/M

CP/M war ja der Vorläufer zu MS-DOS (Bill Gates hat da scheinbar einiges geklaut). Die Grundfunktionen des CP/M-Betriebssystems sind im Floppy-ROM der DDI1-Floppy enthalten, ausserdem ist da ja schon der 8086 drin (dessen Nachfolger der 286er war) der als Controller-Chip missbraucht wird.

Der 8086-Registersatz (so nebenbei)

8Bit		**16Bit**	**Index-Reg.**	**Segment-Reg.**
AH, AL	**--->**	**AX**	**BP (Bytepointer)**	**CS**
BH, BL	**--->**	**BX**	**SP (Stackpointer)**	**DS**
CH, CL	**--->**	**CX**	**SI**	**ES**
DH, DL	**--->**	**DX**	**DI**	**SS (Stack)**

Zur Speicheraufteilung des RAM in CP/M :

```
&BE80  -------------------------------------
       !          Jumps / BIOS             !
&AD00  -------------------------------------
       !              BDOS                 !
&9F00  -------------------------------------
       !              CCP                  !
&9700  -------------------------------------
       !   TPA (Transient Program Area)    !
&0533  ! -   -   -   -   -   -   -   -   - !
       !             Jumps                 !
&0500  ! -   -   -   -   -   -   -   -   - !
       !                                   !
&0100  -------------------------------------
       !         Systemvariablen           !
&0000  -------------------------------------
       !       Bildschirm / AmsDOS         !
&C000  -------------------------------------
```

CCP : (Console Command Processor) : die Benutzeroberfläche von CP/M, enthält residente Befehle und ermöglicht Aufruf von transienten (auf Disk).

BDOS: BASIC Disk Operating System : bildet zusammen mit dem CCP den hardware-unabhängigen Teil von CP/M und stellt die Ein-/Ausgabefunktionen und die Dateiverwaltung zur Verfügung.

BIOS: BASIC Input/Output System : ist der hardware-abhängige Teil von CP/M und ermöglicht dem BDOS die nötigen Optionen.

Die Systemvariablen :

&0000 : Jump zum BIOS Warmboot
&0003 : IO-Byte; über diese Bitmaske werden die physikalischen Geräte den logischen zugeordnet.
&0004 : Drive/User-Default-Einstellung ; Bit 0-4 : Laufwerk, Bit 5-7 : User
&0005 : Jump in BDOS (Haupteinsprung für BDOS-Funktionen) :
Eingangswerte : C=Funktionsnummer, E=Parameter
Ausgangswerte : HL=16Bit-Ausgangswert, A=8Bit-Ausgangswert

&005C : Erster FCB (FileControlBlock) von aufgerufenen Programmen von Disk
&006C : Zweiter FCB (falls vorhanden)
&0080 : Default DMA-Bereich ; in &0080 steht die Länge der Eingabezeile, die ab &0081 folgt.

"Default-Werte" = voreingestellte Standardwerte.

Die BDOS-Funktionen (JP &0005)

00 = System Reset	**21 = sequenziell schreiben**
01 = Konsoleneingabe	**22 = Datei erzeugen**
02 = Konsolenausgabe	**23 = Datei umbenennen**
03 = Reader-Eingabe	**24 = Login-Adresse holen**
04 = Punch-Ausgabe	**25 = Default-Laufwerk holen**
05 = List-Ausgabe	**26 = DMA-Adresse setzen**
06 = Direkte Konsolein-/ausgabe	**27 = Belegungsverz. holen**
07 = I/O-Byte auslesen	**28 = R/O-Status setzen**
08 = I/O-Byte schreiben	**29 = R/O-Status holen**
09 = String ausgeben	**30 = Dateiattribute setzen**
10 = Konsolpuffer lesen	**31 = Diskparameter holen**
11 = Konsolstatus lesen	**32 = Usernummer setzen / holen**
12 = Versionsnummer lesen	**33 = Sektor lesen**
13 = Disksystem Reset	**34 = Sektor schreiben**
14 = Default-Laufwerk wählen	**35 = Dateigrösse berechnen**
15 = Datei öffnen	**36 = Block definieren**
16 = Datei schliessen	**37 = Laufwerk Reset**
17 = DIR nach 1. Eintrag suchen	**38 = unbenutzt**
18 = DIR nach nächst. Eintr. su.	**39 = unbenutzt**
19 = Datei löschen	**40 = Sektor schreiben, leere Blöcke mit Null füllen.**
20 = sequenziell lesen	

Die BIOS-Einsprungtabelle

```
-----------------------------------------------------------------------------
 &AD00 : BOOT      - CP/M komplett laden
 &AD03 : Warm-Boot - BDOS und CCP laden
 &AD06 : CONST     - Wenn Taste gedrückt wurde ist Akku 0 ansonsten &FF
 &AD09 : CONOUT    - Zeichen im C-Register ausgeben
 &AD0F : LIST      - Zeichen im C-Register an den Drucker senden
 &AD12 : PUNCH     - Zeichen im C-Register and V.24 senden
 &AD15 : READER    - Zeichen von V.24 in A lesen
 &AD18 : HOME      - Lese-/Schreibkopf auf Spur 0 setzen
 &AD1B : SELDSK    - Laufwerk selektieren (in C), HL=0 wenn nicht existent
 &AD1E : SETTRK    - Lese-/Schreibkopf auf Spur in C-Register setzen
 &AD21 : SETSEC    - Sektornummer im C-Register auswählen
 &AD24 : SETDMA    - DMA-Adresse in BC-Register setzen
 &AD27 : READ      - Record (128 Byte) nach DMA lesen, A=0 wenn erfolgreich
 &AD2A : WRITE     - Record schreiben, wenn C=0 normale Operation, wen
                     C=1 Directory-Zugriff, wenn C=2 Block ist leer (Record
                     schreiben ohne Rücksicht auf andere 3 Records im Sektor)
                     A=0 wenn erfolgreich
 &AD2D : LISTST    - A=0 Drucker offline, A=&FF Drucker online
 &AD30 : TRANS     - HL=BC
 &AD33 : SYSTEM    - JUMP zu Firmware-Adr. (z.B. CALL &AD33, DW &BB18)
-----------------------------------------------------------------------------
```

Unter CP/M existieren vier logische (virtuelle) Geräte, CON: (Console), RDR: (Reader), PUN: (Puncher) und LST: (Drucker). Jedem logischen Gerät wird ein physikalisches zugeordnet, was auch im CP/M-Setup verändert werden kann.

IO-Byte	Zuordnung	Bezeichnung
xxxxxx00	CON:=TTY:	SIO Kanal A
xxxxxx01	CON:=CRT:	Tastatur und Bildschirm
xxxxxx10	CON:=BAT:	RDR: Eingabe, LST: Ausgabe
xxxxxx11	CON:=UC1:	SIO Kanal B
xxxx00xx	RDR:=TTY:	SIO Kanal A
xxxx01xx	RDR:=PTR:	EOF Eingabe
xxxx10xx	RDR:=UR1:	SIO Kanal B
xxxx11xx	RDR:=UR2:	Tastatur
xx00xxxx	PUN:=TTY:	SIO Kanal A
xx01xxxx	PUN:=PTP:	Kurzgeschlossene Ausgabe
xx10xxxx	PUN:=UP1:	SIO Kanal B
xx11xxxx	PUN:=UP2:	Bildschirm
00xxxxxx	LST:=TTY:	SIO Kanal A
01xxxxxx	LST:=CRT:	Bildschirm
10xxxxxx	LST:=LPT:	Centronics-Port
11xxxxxx	LST:=UL1:	SIO Kanal B

Zum CP/M Diskettenformat :
COM-Dateien sind binäre Dateien, die im ASCII-Format abgespeichert wurden. Wie wir aus Kapitel 3.6 erfahren können, fängt die Directory-Spur erst ab Track 2 an, weil CP/M die ersten beiden Spuren benutzt. Ist eine Diskette im CP/M-Format formatiert, enthält jedoch keine CP/M-Spuren, so spricht man vom Vendor-Format. Alle Informationen des CP/M-Setups ist werden auch in den CP/M-Spuren abgelegt. Das CP/M-Betriebssystem arbeitet mit einem sogenannten FCB (File Control Block), der alle Informationen enthält, die vom BDOS zur Verarbeitung benötigt. Da unter CP/M beliebig ja viele Dateien gleichzeitig geöffnet sein können, muss für jede Datei ein FCB existieren, welcher folgendermassen aufgebaut ist :

Byte	**Name**	**Bedeutung**
00	**dr**	**Laufwerksnummer (0-16)** **0=default, 1=LW A, 2=LW B, 3=LW C,...**
01-08	**f1-f8**	**Dateiname in Grossbuchstaben**
09-11	**t1-t3**	**Extension in Grossbuchstaben** **(Bit 6+7 f. Dateiatribute, s. Kap. 3.6)**
12	**ex**	**Aktuelle Extensions-Nummer (Folgenr.)**
13	**--**	**reserviert**
14	**--**	**reserviert**
15	**rc**	**Blockzähler f. Extension**
16-31	**d0-dn**	**Blocknummern**
32	**cr**	**Optionaler Anhang d. wahlfreien Zugriff**

Laufwerk-Header

Byte	**Bedeutung**
00/01	**Übersetzungssektor**
03-08	**RAM-Speicher f. BDOS**
09/10	**Directory-Puffer f. alle Laufwerke**
11/12	**Disc-Parameter Block**
13/14	**Prüfung auf Diskwechesel (Login)**
15/16	**Disc Allocationsvektor**

5. Anhang

5.1. Binär-Arithmetik

Hier soll nocheinmal zur Erinnerung einiges zur Binär-Arithmetik und zu Zahlensystemen gesagt sein. Machen wir es kurz und schmerslos...

Ein Byte hat 8 Bit, ein Bit kann den Wert 0 oder 1 haben. Dabei nennt man die letzte (rechteste) Stelle der Binärzahl Bit0 und die erste Stelle Bit7. Das Zahlensystem baut auf Zweierpotenzen auf, wobei Bit0 mit 2 hoch 0, Bit1 mit 2 hoch 1, Bit2 mir 2 hoch 2 ... Bit7 mit 2 hoch 7 mal dem entsprechenden Bitwert in den Gesamtwert einfliesst.

```
Beispiel : 0      0      1      1      0      1      0      0
           0*2^7+ 0*2^6+ 1*2^5+ 1*2^4+ 0*2^3+ 1*2^2+ 0*2^1+ 0*2^0 = 52
```

Daraus ergibt sich für ein Byte der Wertebereich 0 bis 255. Alle Grundrechen-operationen (mal, geteilt, plus, minus) funktionieren beim Binärsystem genauso wie beim Dezimalsystem. Darüberhinaus gibt es die primären Bitverknüpfungen AND, OR, XOR und NOT.

Die NOT-Verknüpfung : macht aus einer 1 eine 0 und umgekehrt (Negation).

```
Beispielsweise wird 00110100
                 zu 11001011.
```

Das Ergebnis nennt man **1er-Komplement.** Um bei Bedarf auch negative Zahlen zur Verfügung zu haben, kann von dem 1er-Komplement genau 1 abgezogen werden, sodass

```
                    11001011
                  - 00000001
                    --------
                    11001010 ergibt,
```

und man erhält die negative Zahl, die man auch **2er-Komplement** nennt (in ASM gibt es einen Befehl dafür namens NEG) . Die ursprüngliche Zahl

```
                    00110100
                  + 11001010 (dem 2er-Komplement)
                    --------
  ergibt also genau 00000000, wie gewohnt.
```

Es gibt also zwei inverse Elemente - das logisch-inverse (1er-Komplement) und das algebraisch-inverse (2er-Komplement). Bei der Verwendung von negativen Zahlen wird der natürliche Zahlenbereich 0, 1..255 in den ganzzahligen Zahlenbereich -127 bis +128 geteilt.

Die **AND-Verknüpfung** : (Zwei Bits an gleicher Stelle gleich? Ergibt 1)

```
Beispiel :          00101011
                AND 01101101
                    --------
             ergibt 10111001
```

Die OR-Verknüpfung : (Eines von zwei Bits an gleicher Stelle 1? Ergibt 1)

```
Beispiel :          01001011
                 OR 01101100
                    --------
             ergibt 01101100
```

Die XOR-Verknüpfung : (NUR EINES von 2 Bits an gleicher Stelle 1? Ergibt 1)

```
Beispiel :          01001011
                XOR 01101100
                    --------
             ergibt 00100100
```

Darüberhinaus gibt es noch die NOR und die NAND-Verknüpfung, wo das Ergebnis der OR- bzw. AND-Verknüpfung negiert wird. Angemerkt werden soll noch, dass eine Negation der Eingangs- UND Ausgangswerte einer AND-Verknüpfung eine OR-Verknüpfung ergibt und umgekehrt.

Hilfreich zu wissen ist auch, dass man Bit0 bis Bit3 eines Bytes Lownibble nennt und Bit4 bis Bit7 dementsprechend Highnibble. Da wir ja noch mit Zahlensystemen mit anderen Basen zu tun haben, zum Beispiel mit einem mit der Basis 16 (Hexadezimalsystem) und 2^(4-1) aber genau 16 ergibt, ist die Umrechnung vom Hexadezimal- ins Binärsystem dementsprechend einfach :

```
                  (hi-nibble)  (low-nibble)

         Binär:        0010      0110
                       \  /      \  /
         Hex.  :        2         8            --> 00100110 = &28
```

Und zuguterletzt noch eine einfache Tatsache, mit der man sich oft Mühen ersparen kann : Schieben nach links kommt der Teilung durch 2 einer Zahl gleich, dementsprechend bewirkt das Schieben nach rechts eine Multiplikation mit 2.

5.2. Register und Flags des Z80

Wie wir ja wissen, ist der Z80-Prozessor die CPU des CPCs. Er hat einen sogenannten doppelten Registersatz. Der Akku (A-Register) ist eines der meistbenutzten Register. Er kann einzeln bearbeitet werden, es gibt aber auch 16-Bit Befehle, die sich auf A mit F (dem Flagregister) zusammengefasst beziehen. Die 16-Bit Register HL,DE und BC können jeweils in 8-Bit-Register gespalten werden. Ausserhalb dieses soeben beschriebenen zentralen Registersatzes, der doppelt vorhanden ist, existieren noch einige andere Register, auf die wir gleich näher eingehen.

```
Zentraler Register-Satz
-----------------------       SP : Stackpointer (Stapelzeiger)
!    A|F   !  A'|F'   !       IX : Implizit-Register X
!    B|C   !  B'|C'   !       IY : Implizit-Register Y
!    D|E   !  D'|E'   !      [PC]: Program Counter (intern)
!    H|L   !  H'|L'   !      I|R : Interrupt / Refresh-Register
-----------------------
```

Der Stackpointer (SP) zeigt auf eine Position des Stapels, welchen wir im nächsten Kapitel ausführlich behandeln. Der Program-Counter (PC) zeigt auf die aktuelle Adresse im Programm, die gerade bearbeitet wird. Er kann nicht direkt durch einen Befehl gelesen oder beschrieben werden, man kann ihn jedoch durch den Stapel beeinflussen. Das Interrupt-Register (I) und das Refresh-Register (R) hängen mit dem Hardware-Interrupt zusammen.

Das Flag-Register F enthält 8 wichtige Bits, die Flags genannt werden. Sie dienen sozusagen als Kennzeichen (Flagge) für bestimmte Ereignisse beim arithmetischen Umgang mit Zahlen. Diese Flags sind dem F-Register wiefolgt zugeordnet :

F-Reg.	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Flag	S	Z	-	H	-	P/V	N	C

Name	Funktion
S : Sign (Vorzeichen)	gesetzt (1) wenn Ergebnis einer Addition oder Subtraktion grösser als 127 ist.
Z : Zero (Null)	gesetzt, wenn Ergebnis einer Subtraktion 0 oder wenn Gleichheit vorliegt.
H : Halbübertrag	wird vom Z80A benutzt
P/V: Parität / Überlauf	Ist gesetzt, wenn ein Überlauf eintritt, zeigt gleichzeitig die Parität eines Bytes an.
N : Subtraktion	gesetzt, wenn das Ergebnis nach einer Subtraktion grösser als 127 (negativ) ist.
C : Carry (Übertrag)	gesetzt, wenn bei einer Addition oder Subtraktion ein Übertrag auftritt.

5.3. Der Stapel

Der Stapel (Stack) besteht zunächst aus einem Adressbereich, und einem Zeiger, dem Stackpointer (SP-Register). Ein Wert eines beliebigen 16-Bit-Registers des zentralen Registersatzes kann mit dem Befehl PUSH auf dem Stapel abgelegt und mit dem Befehl POP wieder geholt werden. Aber Achtung! Man hat auf dem Stapel normalerweise keinen wahlfreien Zugriff. Dies tut man nur in Ausnahmefällen, denn bevor der Sinn des Stapels total verfehlt ist, indem man den Stackpointer an einer Adresse zwischenspeichert, ihn auf den gewünschten Wert setzt, diesen holt und den Stackpointer wieder an seinen alten Platz - 1 setzt, legt man lieber einen eigenes Datenfeld an, das ist weniger umständlich und ist sicherer. Denn auf dem Stapel werden auch Abprungadressen vermerkt, z.B. von wo aus durch einen CALL gesprungen wurde, damit nachvollzogen werden kann, wo das Programm weiterlaufen soll, nachdem die Unterroutine mit RET beendet ist. Deshalb sollte der Stackpointer auch nicht unbedacht verändert werden, dies kann zum Absturz führen.

```
<wert-n>  <-- Stackpointer (SP)
   .
   .
   .
<wert-3>
<wert-2>
<wert-1>
```

Zu beachten ist die Reihenfolge. Legt man die Werte HL,DE und BC auf dem Stapel ab mit :

```
PUSH HL
PUSH DE
PUSH BC
```

und will sie später wieder vom Stapel holen, muss man dies in umgekehrter Reihenfolge tun, um die gleichen Werte wieder in den gleichen Registern zu haben, weil der Wert des BC-Registers nun "oben" liegt, mit :

```
POP BC
POP DE
POP HL
```

Die Befehle PUSH HL, POP BC bewirken zum Beispiel, das der Wert des HL-Registers auf den Stapel gelegt wird und anschliessend ins BC-Register zurückgeholt wird, dies kann also als "LD BC,HL" aufgefasst werden, einen Befehl, der KEIN Befehl des Z80-Befehlssatzes ist, also nicht wirklich existiert und trotzdem mit einem Trick realisiert wird.

Da nun auch Sprungadressen auf dem Stapel gespeichert werden, ist es z.B. nicht ohne weiteres möglich, im Hauptprogramm ein Register auf den Stapel zu legen und an ein Unterprogramm zu verzweigen, welches sich diesen Wert mit POP einfach wieder holt. In folgendem Programm :

```
          LD HL,&C000
          PUSH HL
          CALL @sub
             .
             .
          RET
    @sub  POP HL
             .
             .
```

wird also die Unterroutine @sub NICHT den Wert &C000 vom Stapel holen, sondern die Absprungadresse des CALLs. Dies führt aber wiederrum dazu, dass nun die Adresse &C000 als Absprungadresse gilt, was in 99% aller Fälle unerwünschte Folgen hat. Wenn man nun genau weiss, wie oft verschachtelt wurde, kann man sich den Wert holen. Für unser Beispiel gilt dann :

```
         LD HL,&C000
         PUSH HL
         CALL @sub
             .
             .
         RET
   @sub  POP DE       ; Absprungadresse vom Stapel in DE
         POP HL       ; HL vom Stapel holen
         PUSH DE      ; Absprungadresse wieder auf den Stapel
             .
             .
```

Weiterhin haben z.B. die Befehle LD HL,&A000, PUSH HL, RET sowohl als Unterprogramm (mit CALL aufgerufen) als auch als Hauptprogramm (Mit JP aufgerufen) die gleiche Wirkung wie ein JP &A000, denn diese Adresse wird auf den Stapel gelegt und durch das RET als nächste Einsprungadresse dem PC übergeben. So kann man also auch das PC-Register indirekt schreiben und lesen. Ein Beispiel hierzu liefert die Adress-Verteilerroutine im Floppy-ROM (siehe Kapitel DDI1-Floppy).

5.4. Speicheraufteilung

```
&0000 -------- -----------
      ! 16K  ! !  16K    !
      ! RAM  ! !  ROM    !
      !      ! !Betr.sys.!
&4000 -------- -----------
      ! 16K  !
      ! RAM  !
&8000 -------- ----------- ----------
      ! 16K  ! !  16K    ! !  16K   !
      ! RAM  ! !  ROM    ! !max. 252!
      !      ! ! BASIC   ! !Ext. ROM!
&C000 -------- ----------- ----------
```

5.5. RST's

RST steht für RESTART und bedeutet "Neustart". Die RST-Befehle werden zum grössten Teil dazu verwendet, um in andere ROMs zu springen. Dabei wird bei ihrem Aufruf an die jeweilige Adresse (&0000 - &0038) verzweigt wo dann die entsprechenden Routinen / Jumps stehen, die das ensprechende erledigen.

```
Befehl Source-  Name         Sprung zu    Beschreibung / Anwendung
        Code                 Adresse


RST0    &C7     Reset        &0000        Führt einen Reset aus (Call 0)

RST1    &CF     Low Jump     &0008        RST1 (RST&08)
                                          DW <adresse>
     (Ab &0008 Sprung nach  &B982)        DB <maske> -->Bit14 : 0 = Betr.sys.
                                                                1 = RAM
                                                        Bit15 : 0 = BASIC-ROM
                                                                1 = RAM

RST2    &D7     Side Call    &0010        RST2 (RST&10)
                                          DW <adresse>
     (Ab &0010 Sprung nach  &BA16)        DB <maske> -->Bit14+Bit15 : Romnr.

RST3    &DF     Far Call     &0018             RST3 (RST&18)
                                               DW TAB
     (Ab &0018 Sprung nach  &B9BF)               .
                                                 .
                                          @TAB DW <adresse>
                                               DB <ROM-Konfiguration>

      Wert für ROM-Konfiguration        &0000-&3FFF   &C000-&FFFF
      ------------------------------------------------------------
                &07                          RAM        Floppy-ROM
                &FC                       Betr.sys.      BASIC-ROM
                &FD                          RAM         BASIC-ROM
                &FE                       Betr.sys.         RAM
                &FF                          RAM            RAM


RST4    &E7     RAM LAM      &0020        LD A,(HL) aus RAM, unabhängig davon
                                          welche ROMs gerade ein-/ausgeschaltet
     (Ab &0020 Sprung nach  &BACB)        sind.

RST5    &EF    Firm Jump     &0028        RST5 (RST&28) ; Sprung ins Betr.sys.
                                          DW <adresse>

RST6    &F7    User Restart &0030         Vom User frei belegbar. Wird der
                                          Source-Code &F7 im Programm gefunden,
     (Ab &0030 Sprung zu User-Routine)    wird nach &0030 gesprungen, wo dann
                                          ein JP zu einer Anwenderroutine
                                          stehen kann (z.B. f. Breakpoint)

RST7    &FF    Interrupt-    &0038        Interrupt-Steuerung
                 Entry

    (Ab &0038  Sprung nach  &B939)
```

Hier einige nützliche ROM-Ansprünge :

ROM-Nr.	ROM	Adresse	Wirkung
FE	B.sys.	&0888	Jump Restore, erneuert Firmware-Calls (&BB00-&BDCA)
FD	BASIC	&EE79	Zahl in HL wird dezimal ausgegeben
FD	BASIC	&E9BD	BASIC-Befehl RUN
FD	BASIC	&CA94	Fehler-Aussprung (E = Fehlernummer)
FD	BASIC	&DD3F	Blanks überlesen (Eingabepuffer)
FD	BASIC	&DD37	Zeichen prüfen (entspr. Zeichen direkt n. Far Call)
FD	BASIC	&DD86	Variable suchen (DE=adr, C=Typ)
FD	BASIC	&C132	BASIC-Befehl CLEAR
FD	BASIC	&C12B	BASIC-Befehl NEW
FD	BASIC	&CB5A	BASIC-Befehl STOP
FD	BASIC	&CBC0	BASIC-Befehl CONT
FD	BASIC	&C064	READY-Modus
FD	BASIC	&E728	BASIC-Befehl DELETE
FD	BASIC	&E10D	BASIC-Befehl LIST (BC bis DE)
FD	BASIC	&FFC4	DE=DE-HL
FD	BASIC	&FFCF	HL=HL-DE
FD	BASIC	&FFDA	BC=HL-DE
FD	BASIC	&FFE7	HL=HL-BC
07	DDI1	&C033	Fehlermeldungen ein/aus (0 = aus)
07	DDI1	&C036	Eingabewert HL=Tabelle : Byte 0/1=Motor-Vorlaufzeit, Byte 2/3=Motor-Nachlaufzeit, Byte 4=Write-off Zeit, Byte 5=Kopf-Plazierungs-Zeit, Byte6=Schrittgeschw., Byte 7=Unload-Delay, Byte 8=&03 (siehe RAM &BE44)
07	DDI1	&C039	Select Format (A = Format-Nr.)
07	DDI1	&C03C	Read Sector \
07	DDI1	&C03F	Write Sector (siehe Kapitel 3.6)
07	DDI1	&C042	Format Track /
07	DDI1	&C045	Kopf plazieren E=Laufwerk, D=Track
07	DDI1	&C048	LW-Status holen (A=Laufwerk); Ausgangswert = A Bit6 = Schreibschutz; Bit5 = Disk geloggt; Bit4 = gesetzt, wenn Kopf auf Spur 0
07	DDI1	&C04B	Anzahl der Leseversuche einstellen (A=Anzahl)
07	DDI1	&C56C	Sektor lesen (E=LW-Nr.), andere Werte in &BE4F-&BE52
07	DDI1	&CA72	A <--> (&BE78) Flag f. Fehlermeldung
07	DDI1	&C603	A <--> (&BE66) Flag f. Disk-Leseversuche
07	DDI1	&CE14	Disk-Login
07	DDI1	&D683	Allokations-Tabelle anlegen
07	DDI1	&D698	Suche Directory-Eintrag Eingabeparameter : B = String Ausgabeparameter : DE = Eintrag, HL=Nr.
07	DDI1	&D97A	Schreibe Directory-Eintrag

```
ROM-Nr.   ROM     Adresse  Wirkung
------------------------------------------------------------------------------

  07      DDI1    &C1BC     (CPM OUT)    (&C006)
  07      DDI1    &C1B2     |CPM         (&C009) \
  07      DDI1    &CCD1     |DISC        (&C00C)  \
  07      DDI1    &CCD5     |DISC.IN     (&C00F)   !
  07      DDI1    &CCE4     |DISC.OUT    (&C012)   !
  07      DDI1    &CCFD     |TAPE        (&C015)   !
  07      DDI1    &CD01     |TAPE.IN     (&C019)   !
  07      DDI1    &CD18     |TAPE.OUT    (&C01B)    \
  07      DDI1    &CDDA     |A           (&C01E)    /  Floppy-RSX
  07      DDI1    &CDDD     |B           (&C021)   !
  07      DDI1    &CDE4     |DRIVE       (&C024)   !
  07      DDI1    &CDFE     |USER        (&C027)   !
  07      DDI1    &D42E     |DIR         (&C02A)   !
  07      DDI1    &D48A     |ERA         (&C02D)  /
  07      DDI1    &D4C4     |REN         (&C030) /



  07      DDI1    &D513     = Call &BC9B \
  07      DDI1    &D0D8     = Call &BC98  \
  07      DDI1    &D08F     = Call &BC95   !
  07      DDI1    &D1C2     = Call &BC92   !
  07      DDI1    &D1D8     = Call &BC8F   !
  07      DDI1    &CF37     = Call &BC8C    \ Für Floppy
  07      DDI1    &D065     = Call &BC89    /
  07      DDI1    &D069     = Call &BC86   !
  07      DDI1    &CFF5     = Call &BC83   !
  07      DDI1    &CF64     = Call &BC80   !
  07      DDI1    &D1BC     = Call &BC7D   !
  07      DDI1    &D1B6     = Call &BC7A  /
  07      DDI1    &CEAF     = Call &BC77 /
```

5.6. Der Z80-Befehlssatz

Man kann den Z80-Befehlssatz in drei grosse Tabellen einteilen, wobei Tabelle 1 die elementarsten Befehle enthält (&00 - &FF), die zweite Tabelle enthält die Schiebe- und Bit-Befehle zur Manipulation einzelner Bits und wird mit dem Source-Code &CB angesprochen. Die dritte Tabelle wird mit dem Sourcecode &ED aktiviert (d.h. das nachfolgende Byte bezieht sich nun auf die dritte Tabelle) und enthält zum grössten Teil die Port-Befehle, aber auch noch einige erweiterte Lade- und Blocktransfer-Befehle. Die Bytes &DD und &FD zeigen an, dass der nachfolgende Befehl, der sich normalerweise auf das HL-Register bezieht, nun für das IX-Register (&DD) bzw. das IY-Register (&FD) gelten soll.

5.6.1. Funktionen der Befehle

Befehl	**Bedeutung**
LD <a>,<b>	Kopiere <b> nach <a>
PUSH	Erhöht SP und legt Wert des Registers auf den Stapel
POP	Kopiert Wert vom Stapel in Register und erniedrigt SP
EX r1 , r2	Vertauscht Register r1 mit Register r2
EXX	Vertauscht AF,BC,DE,HL mit AF',BC',DE',HL' (zentr. Reg.satz)
JP [<bed>,]nn	Sprung zu Adresse nn, eventuell mit Bedingung <bed>
JR r	Relativer Sprung mit Sprungweite r r < 128 vorwärts (Ab erstem Byte nach r) r >=128 rückwärts (Ab erstem Byte nach JR)
CALL	Aufruf einer Unterroutine
DJNZ	Multibefehl : DEC B; CP 0; JR NZ,r
RST	(siehe Kapitel 5.4)
LDI	Multibefehl : LD A,(HL); LD (DE),A; INC HL; INC DE;DEC BC; LD A,B; OR C
LDIR	Wie oben, nur wird solange wiederholt, bis BC = 0 ist.
LDD	Multibefehl : LD A,(HL); LD (DE),A; DEC HL; DEC DE;DEC BC; LD A,B; OR C
LDDR	Wie open, nur wird solange wiederholt, bis BC = 0 ist.
CPI	Multibefehl : CP (HL); INC HL; DEC BC
CPIR	Wie oben, nur wird solange wiederholt, bis BC = 0 oder A=(HL).
CPD	Multibefehl : CP (HL); DEC HL; DEC BC
CPDR	Wie oben, nur wird solange wiederholt, bis BC = 0 oder a=(HL).
NOP	No OPeration (keine Wirkung)
HALT	Das Programm wird bis zum nächsten Interrupt angehalten
CCF	Das Carry-Flag wird negiert
SCF	Das Carry-Flag wird auf 0 gesetzt
EI	Interrupts entsperren
DI	Interrupts sperren
IM0	Interrupt-Mode 0 : Prozessor wartet nach Interrupt auf den Befehl eines externen Gerätes.
IM1	Interrupt-Mode 1 : Standardeinstellung, verzweigt bei Interrupt
IM2	Interrupt-Mode 2 : (Vektorinterrupt) - Es wird an eine vom Interrupt abhängige Adresse verzweigt, die in einer Tabelle steht.
TSTI	Interrupt testen
OUT	Port schreiben

IN	Port lesen
INI	Multibefehl : IN A,(C); LD (HL),A; INC HL; DEC B
INIR	Wie oben, nur wird solange wiederholt, bis BC = 0 ist.
IND	Multibefehl : IN A,(C); LD (HL),A; DEC HL; DEC B
INDR	Wie oben, nur wird solange wiederholt, bis BC = 0 ist.
OUTI	Multibefehl : LD A,(HL); OUT (C),A; INC HL; DEC B
OUTIR	Wie oben, nur wird solange wiederholt, bis BC = 0 ist.
OUTD	Multibefehl : LD A,(HL); OUT (C),A; DEC HL; DEC B
OUTDR	Wie oben, nur wird solange wiederholt, bis BC = 0 ist.

BIT	Bit vergleichen. Wenn Bit = 0 --> Z-Flag = 1
RES	Bit zurücksetzen (0)
SET	Bit setzen (1)

ADD	Addieren
ADC	Addieren, Übertrag in Carry
SUB	Subtrahieren
SBC	Subtrahieren, Übertrag in Carry
AND	AND-Verknüpfung
OR	OR-Verknüpfung
XOR	XOR-Verknüpfung
CP	Vergleich.

C	Z	
0	0	A > n
0	1	A = n
1	0	A < n

INC	Erhöhen um 1 (incrementieren)
DEC	Erniedrigen um 1 (decrementieren)
CPL	Akku logisch negieren (1er-Komplement)
NEG	Akku algebraisch negieren (2er-Komplement)
DAA	Dezimal-Anpassung des Akkus (A-Register).

Oft werden Zahlen Ziffer für Ziffer abgespeichert. Dabei wird pro Ziffer ein Nibble (4 Bits) belegt, wobei dann noch sechs Bitkombinationen frei sind, die nicht benutzt werden dürfen, da es nur zehn Ziffern gibt. Beim Addieren oder Subtrahieren kann sowas aber passieren, der DAA-Befehl dient dazu, den Akku so zu verändern, dass die Ziffern nachher wieder stimmen, indem ein entsprechender Wert addiert wird.

Bit 4-7	Bit 0-3	zu A addiert	Carry-Flag	N-Flag
0-9	0-9	00	0	0
0-8	A-F	06	0	0
0-9	A-F	66	1	0
A-F	0-9	60	1	0
A-F	A-F	66	1	0

RR	Carry --> 76543210 --> Carry
RL	Carry <-- 76543210 <-- Carry
RRC	Bit0 --> 76543210 --> Carry
RLC	Carry <-- 76543210 <-- Bit7
SRA	Bit7 --> 76543210 --> Carry
SLA	Carry <-- 76543210 <-- 0
SRL	0 --> 76543210 --> Carry
RRCA	= RRC A
RLCA	= RLC A
RRA	= RR A
RLA	= RL A
RLD	3210 (A-Reg.) <-- 76543210 (HL-Reg.) <-- Bit3 (A-Reg.)
RRD	3210 (A-Reg.) --> 76543210 (HL-Reg.) --> Bit3 (A-Reg.)

5.6.2. Die Befehlstabelle

Nun zur Zuordnung der Befehle zum entsprechenden OP-Code. Aus Platzgründen wird hier der Befehlssatz jedoch nicht nach OP-Codes, sondern nach Befehlen sortiert aufgeführt. Dies erschwert zwar ein "disassemblieren" von Hand, hilft aber dafür um so mehr dabei, einen Disassembler zu schreiben, da hier die Systeme erkennbarer werden, die oft in der Befehlszuordnung verwendet wurden, und die durch ihre Berechenbarkeit das Programmieren einer Tabelle ersparen.

Legende zum lesen der Tabellen :

Im Befehl	Im OP-Code	Bedeutung
n	**n1**	beliebiger 8-Bit-Wert
nn	**n1n2**	beliebiger 16-Bit-Wert
(nn)	**n1n2**	bezieht sich auf den Speicherinhalt mit der Adresse nn, mit n1 als High-Byte, n2 als Lowbyte
r	**n1**	ein Wert zwischen -127 und +128 für relative Sprünge, enthält die Anzahl der Bytes, von der momentanen Adresse aus vor- bzw. zurückgesprungen werden soll.
d	**dd**	bezeichnet einen Zähler für das IX- und IY-Register, der zu dem Adresswert des jeweiligen Registers dazuaddiert wird, also zusätzlich zur impliziten Adressierung verwendet werden kann. Dies ist (wohl gemerkt!) nur mit diesen beiden Registern möglich, weshalb sie sich besonders als Pointer für eigene Stacks, Heaps, etc. eignen.
rr	--	steht für ein beliebiges 16-Bit-Register
(rr)	--	bezieht sich auf den Inhalt des entspr. Registers

Für die Flag-Beeinflussung durch Befehle gelten die Zeichen :

0 nicht gesetzt
1 gesetzt
- unverändert
* operationsabhängig
? unbestimmt

8-Bit-Ladebefehle :

	A	**B**	**C**	**D**	**E**	**H**	**L**	**(HL)**	**(DE)**	**(BC)**	**(nn)**	**n**
LD A,..	**7F**	**78**	**79**	**7A**	**7B**	**7C**	**7D**	**7E**	**1A**	**0A**	**3An1n2**	**3En1**
LD B,..	**47**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	--	--	--	**0En1**
LD C,..	**4F**	**48**	**49**	**4A**	**4B**	**4C**	**4D**	**4E**	--	--	--	**16n1**
LD D,..	**57**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	--	--	--	**1En1**
LD E,..	**5F**	**58**	**59**	**5A**	**5B**	**5C**	**5D**	**5E**	--	--	--	**26n1**
LD H,..	**67**	**60**	**61**	**62**	**63**	**64**	**65**	**66**	--	--	--	**2En1**
LD L,..	**6F**	**68**	**69**	**6A**	**6B**	**6C**	**6D**	**6E**	--	--	--	**36n1**
LD (HL),..	**77**	**70**	**71**	**72**	**73**	**74**	**75**	--	--	--	--	--
LD (DE),..	**12**	--	--	--	--	--	--	--	--	--	--	--
LD (BC),..	**02**	--	--	--	--	--	--	--	--	--	--	--
LD (nn),..	**32n1n2**	--	--	--	--	--	--	--	--	--	--	--

	A	B	C	D	E	H	L	n
LD ..,(IX+d)	DD7Edd	DD46dd	DD4Edd	DD56dd	DD5Edd	DD66dd	DD6Edd	DD36ddnn
LD ..,(IY+d)	FD7Edd	FD46dd	FD4Edd	FD56dd	FD5Edd	FD66dd	FD6Edd	FD36ddnn
LD (IX+d),..	DD77dd	DD70dd	DD71dd	DD72dd	DD73dd	DD74dd	DD75dd	--
LD (IY+d),..	FD77dd	FD70dd	FD71dd	FD72dd	FD73dd	FD74dd	FD75dd	--

Flags :		S	Z	H	P/V	N	C (F-Register)
LD A,I	ED57	*	*	0	*	0	-
LD A,R	ED5F	*	*	0	*	0	-
LD I,A	ED47	-	-	-	-	-	-
LD R,A	ED4F	-	-	-	-	-	-

16-Bit-Ladebefehle :

	BC	DE	HL	SP	IX	IY
LD ..,nn	01n1n2	11n1n2	21n1n2	31n1n2	DD21n1n2	FD21n1n2
LD ..,(nn)	ED4Bn1n2	ED5Bn1n2	2An1n2	ED7Bn1n2	DD2An1n2	FD2An1n2
LD (nn),..	ED43n1n2	ED53n1n2	22n1n2	ED73n1n2	DD22n1n2	FD22n1n2
LD SP,	--	--	F9	--	DDF9	FDF9
PUSH...	C5	D5	E5	F5	DDE5	FDE5
POP...	C1	D1	E1	F1	DDE1	FDF1

EX (SP),HL	E3
EX (SP),IX	DDE3
EX (SP),IY	FDE3
EX DE,HL	EB
EX AF,AF'	08
EXX	D9

8-Bit-Arithmetik-Befehle

	B	C	D	E	H	L	(HL)	A	n	(IX+d)	(IY+d)	S	Z	H	P/V	N	C
ADD	80	81	82	83	84	85	86	87	C6n1	DD86dd	FD86dd	*	*	*	*	0	*
ADC	88	89	8A	8B	8C	8D	8E	8F	CEn1	DD8Edd	FD8Edd	*	*	*	*	0	*
SUB	90	91	92	93	94	95	96	97	D6n1	DD96dd	FD96dd	*	*	*	*	1	*
SBC	98	99	9A	9B	9C	9D	9E	9F	DEn1	DD9Edd	FD9Edd	*	*	*	*	1	*
AND	A0	A1	A2	A3	A4	A5	A6	A7	E6n1	DDA6dd	FDA6dd	*	*	1	*	0	0
XOR	A8	A9	AA	AB	AC	AD	AE	AF	EEn1	DDAEdd	FDAEdd	*	*	1	*	0	0
OR	B0	B1	B2	B3	B4	B5	B6	B7	F6n1	DDB6dd	FDB6dd	*	*	1	*	0	0
CP	B8	B9	BA	BB	BC	BD	BE	BF	FEn1	DDBEdd	FDBEdd	*	*	*	*	1	*
INC	04	0C	14	1C	24	2C	34	3C	----	DD34dd	FD34dd	*	*	*	*	0	-
DEC	05	0D	15	1D	25	2D	35	3D	----	DD35dd	FD35dd	*	*	*	*	1	-

		S	Z	H	P/V	N	C
DAA	27	*	*	*	*	-	*
CPL	2F	-	-	1	-	1	-
NEG	ED44	*	*	*	*	1	*

16-Bit-Arithmetik-Befehle

	BC	DE	HL	SP	IX	IY	S	Z	H	P/V	N	C
INC..	03	13	23	33	DD23	FD23	-	-	-	-	-	-
DEC..	0B	1B	2B	3B	DD2B	FD2B	-	-	-	-	-	-
ADD HL,..	09	19	29	39	--	--	-	-	*	-	0	-
ADC HL,..	ED4A	ED5A	ED6A	ED7A	--	--	*	*	*	*	0	*
SBC HL,..	ED42	ED52	ED62	ED72	--	--	*	*	*	*	1	*
ADD IX,..	DD09	DD19	--	DD39	DD29	--	-	-	*	-	0	*
ADD IY,..	FD09	FD29	--	FD39	--	FD29	-	-	*	-	0	*

Sprungbefehle :

	--	Z,	NZ,	C,	NC,	PE,	PO,	M,	P,
JP..nn	C3n1n2	CAn1n2	C2n1n2	DAn1n2	D2n1n2	EAn1n2	E2n1n2	FAn1n2	F2n1n2
CALL..nn	CDn1n2	CCn1n2	C4n1n2	DCn1n2	D4n1n2	ECn1n2	E4n1n2	FCn1n2	F4n1n2
RET..	C9	C8	C0	D8	D0	E8	E0	F8	F0
JR..r	18n1	28n1	20n1	38n1	30n1	--	--	--	--

Z = Springe wenn 0 (Z=1)
NZ = Springe wenn <>0 (Z=0)
C = Springe wenn Übertrag (C=1)
NC = Springe wenn kein Übertrag (C=0)
PO = Springe wenn ungerade Parität (P/V=0)
PE = Springe wenn gerade Parität (P/V=1)
P = Springe wenn positiv (S=1)
M = Springe wenn negativ (S=0)

RST	&00	&08	&10	&18	&20	&28	&30	&38
	C7	CF	D7	DF	E7	EF	F7	FF

DJNZ r 10n1 (DEC B: CP 0: JR NZ,r)
RETI ED4D (Zurück von Interrupt)
RETN ED45 (Zurück von nicht maskierbarem Interrupt)

Rotations- und Schiebebefehle

	B	C	D	E	H	L	(HL)	A	(IX+d)	(IY+d)	S	Z	H	P/V	N	C
RR	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	CB1F	DDCBdd1E	FDCBdd1E	*	*	0	*	0	*
RL	CB10	CB11	CB12	CB13	CB14	CB15	CB16	CB17	DDCBdd16	FDCBdd16	*	*	0	*	0	*
RRC	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	CB0F	DDCBdd0E	FDCBdd0E	*	*	0	*	0	*
RLC	CB00	CB01	CB02	CB03	CB04	CB05	CB06	CB07	DDCBdd06	FDCBdd06	*	*	0	*	0	*
SRA	CB28	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	CB2F	DDCBdd2E	FDCBdd2E	*	*	0	*	0	*
SLA	CB20	CB21	CB22	CB23	CB24	CB25	CB26	CB27	DDCBdd26	FDCBdd26	*	*	0	*	0	*
SRL	CB38	CB39	CB3A	CB3B	CB3C	CB3D	CB3B	CB2F	DDCBdd3E	FDCBdd3E	*	*	0	*	0	*

		S	Z	H	P/V	N	C
RRA	1F	-	-	0	-	0	*
RLA	17	-	-	0	-	0	*
RRCA	0F	-	-	0	-	0	*
RRCA	0F	-	-	0	-	0	*
RLCA	07	-	-	0	-	0	*
RLD	ED6F	*	*	0	*	0	-
RRD	ED67	*	*	0	*	0	-

Bit-Manipulationsbefehle (<Befehl> x,<register>)

	B	C	D	E	H	L	(HL)	A	(IX+d)	(IY+d)	S	Z	H	P/V	N	C
BIT0	CB40	CB41	CB42	CB43	CB44	CB45	CB46	CB47	DDCBdd46	FDCBdd46	?	*	1	?	0	-
BIT1	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E	CB4F	DDCBdd4E	FDCBdd4E	?	*	1	?	0	-
BIT2	CB50	CB51	CB52	CB53	CB54	CB55	CB56	CB57	DDCBdd56	FDCBdd56	?	*	1	?	0	-
BIT3	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E	CB5F	DDCBdd5E	FDCBdd5E	?	*	1	?	0	-
BIT4	CB60	CB61	CB62	CB63	CB64	CB65	CB66	CB67	DDCBdd66	FDCBdd66	?	*	1	?	0	-
BIT5	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E	CB6F	DDCBdd6E	FDCBdd6E	?	*	1	?	0	-
BIT6	CB70	CB71	CB72	CB73	CB74	CB75	CB76	CB77	DDCBdd76	FDCBdd76	?	*	1	?	0	-
BIT7	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E	CB7F	DDCBdd7E	FDCBdd7E	?	*	1	?	0	-
RES0	CB80	CB81	CB82	CB83	CB84	CB85	CB86	CB87	DDCBdd86	FDCBdd86	-	-	-	-	-	-
RES1	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E	CB8F	DDCBdd8E	FDCBdd8E	-	-	-	-	-	-
RES2	CB90	CB91	CB92	CB93	CB94	CB95	CB96	CB97	DDCBdd96	FDCBdd96	-	-	-	-	-	-
RES3	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E	CB9F	DDCBdd9E	FDCBdd9E	-	-	-	-	-	-
RES4	CBA0	CBA1	CBA2	CBA3	CBA4	CBA5	CBA6	CBA7	DDCBddA6	FDCBddA6	-	-	-	-	-	-
RES5	CBA8	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE	CBAF	DDCBddAE	FDCBddAE	-	-	-	-	-	-
RES6	CBB0	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6	CBB7	DDCBddB6	FDCBddB6	-	-	-	-	-	-
RES7	CBB8	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE	CBBF	DDCBddBE	FDCBddBE	-	-	-	-	-	-
SET0	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6	CBC7	DDCBddC6	FDCBddC6	-	-	-	-	-	-
SET1	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE	CBCF	DDCBddCE	FDCBddCE	-	-	-	-	-	-
SET2	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6	CBD7	DDCBddD6	FDCBddD6	-	-	-	-	-	-
SET3	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE	CBDF	DDCBddDE	FDCBddDE	-	-	-	-	-	-
SET4	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6	CBE7	DDCBddE6	FDCBddE6	-	-	-	-	-	-
SET5	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE	CBEF	DDCBddEE	FDCBddEE	-	-	-	-	-	-
SET6	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6	CBF7	DDCBddF6	FDCBddF6	-	-	-	-	-	-
SET7	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE	CBFF	DDCBddFE	FDCBddFE	-	-	-	-	-	-

CPU-Steuerbefehle :

		S	Z	H	P/V	N	C
	Flags :	S	Z	H	P/V	N	C (F-Register)
NOP	00	-	-	-	-	-	-
HALT	76	-	-	-	-	-	-
CCF	3F	-	-	*	-	0	*
SCF	37	-	-	0	-	0	1
EI	FB	-	-	-	-	-	-
DI	F3	-	-	-	-	-	-
IM 0	ED46	-	-	-	-	-	-
IM 1	ED56	-	-	-	-	-	-
IM 2	ED5E	-	-	-	-	-	-
TSTI (c)	ED70	?	?	?	?	?	?

Blocktransfer- und Suchbefehle :

		Flags : S	Z	H	P/V	N	C (F-Register)
LDI	EDA0	-	-	0	*	0	-
LDIR	EDB0	-	-	0	0	0	-
LDD	EDA8	-	-	0	*	0	-
LDDR	EDB0	-	-	0	0	0	-
CPI	EDA1	*	*	*	*	1	-
CPIR	EDB1	*	*	*	*	1	-
CPD	EDA9	*	*	*	*	1	-
CPDR	EDB9	*	*	*	*	1	-

Port-Befehle :

	A	B	C	D	E	H	L	S	Z	H	P/V	N	C
IN ..,(C)	ED78	ED40	ED48	ED50	ED58	ED60	ED68	*	*	0	*	0	-
OUT (C),..	ED79	ED41	ED49	ED51	ED59	ED61	ED69	-	-	-	-	-	-

	S	Z	H	P/V	N	C
IN A,(n)	-	-	-	-	-	-
OUT (n),A	-	-	-	-	-	-
INI	*	*	*	*	1	-
INIR	*	1	*	*	1	-
IND	*	*	*	*	1	-
INDR	*	1	*	*	1	-
OUTI	*	*	*	*	1	-
OUTIR	*	1	*	*	1	-
OUTD	*	*	*	*	1	-
OUTDR	*	1	*	*	1	-

Kleine Trickkiste (Spart Zeit + Bytes)

```
-------------------------
         OR A : C-Flag:= 0
         OR x : CP 0
XOR A, SUB A : A:= 0
CP A, LD A,A : Z-Flag:= 1
          CPL : A XOR &FF
          SCF : C-Flag:= 1
        AND A : H-Flag:= 1
-------------------------
```

5.7. Firmware und reservierter Speicher

Es ist zwar eine sehr lange Liste, aber für einen erfahrenen Anwender die nützlichste Information. Hier folgt nun also eine fast komplette Auflistung des RAM-Speichers und eine Beschreibung, wie die entsprechenden Bereiche benutzt werden. Wer mir noch helfen möchte, die Unbekannten aufzufüllen, dem bin ich sehr dankbar...

I: = Eingangswerte ; O: = Ausgangswerte ; ->&xxxx = Jump zu Adresse &xxxx

5.7.1. Allgemeine Vektoren

```
&0000 RST0 Reset
&0008 RST1 Low Jump      (->&B982)
&000B KL Low Pchl.       (->&B97C)
&000E JP (BC)
&0010 RST2 Low Side Call (->&BA16)
&0013 KL Side Pchl.      (->&BA10)
&0016 JP (DE)
&0018 RST3 Low Far Call  (->&B9BF)
&001B KL Far Pchl.       (->&B9B1)
&001E JP (HL)
&001F frei
&0020 RST4 RAM LAM       (->&BACB)
&0023 KL Far ICall       (->&B9B9)
&0026 frei
&0027 frei
&0028 RST5 Firm Jump     (->&BA2E)
&002B frei \
&002C frei   (ursprünglich f. RST0
&002D frei   nach High HL Restore
&002E frei   im ROM)
&002F frei /
&0030 RST6 User RST (urspr. f. RST0...)
&0038 RST7 Interr. Entry (->&B939)
&003B RET (Ext. Interrupt Patch)
&003E frei
&003F frei

&0040 - &016F Eingabepuffer (Tokens)
&0170 - &A6FF frei verfügbar
```

DDI1-Speicher :

```
&A700 Aktuelles Laufwerk
&A701 Aktuelle User-Nr.
&A702 Flag f. ??
&A703 - &A704 Zeiger auf 1. Laufwerks-Tabelle
&A705 Flag f. ??
&A706 -&A707  ??
&A708 OPENIN-Flag (&FF = geschlossen)
&A709 Bei geöffneter Datai entspr. DIR-Eintrag
&A72C OPENOUT-Flag (&FF = geschlossen)
&A72D Bei geöffneter Datai entspr. DIR-Eintrag
&A750 Flag (&00 = OPENIN, &01 = in char, &02 = in direct)
```

```
&A751 Adresse d. 2K-Puffers f. ASCII oder Start d. aktuellen/letzten Blocks
&A753 Adresse d. nexten Bytes bei ASCII oder Start d. 2K-Puffers f. BAS/BIN
&A755 Bei LOAD/SAVE entspr. 1. Block
&A79A - &A7E3 frei / Puffer
&A7E4 Flag (&00 = OPENOUT, &01 = in char, &02 = in direct)
&A7E5 - &A7F2 ??
&A7F3 - &A863 Puffer
&A864 - &A88A TAPE-Vektoren (zwischengespeichert)
&A88B Einsprungadr. + ROM-Nr. f. Floppy
&A88E ??
&A88F ??
```

2. Laufwerkstabelle

```
Laufwerk A                                         Laufwerk B

  &A890    \  Sektorgrösse (x * 128 Bytes) /  &A8D0
  &A891    /                              \  &A8D1
  &A892      Blockgrösse   (&03=1024 Bytes)   &A8D2
  &A893   Blockgrösse/128-1 (&07=1024 Bytes)  &A8D3
  &A894  Blockgr./1024 (wenn insgesamt <256)   &A8D4
                     sonst /2048-1
  &A895    \        Freie Blocks / KB       /  &A8D5
  &A896    /                                \  &A8D6
  &A897    \         max. Anzahl d.         /  &A8D7
  &A898    /       Directory-Einträge      \  &A8D8
  &A899    \       Bitmasken f. DIR-Block   /  &A8D9
  &A89A    /                                \  &A8DA
  &A89B    \   max. Anzahl Blocks/ extent.  /  &A8DB
  &A89C    /    (=(&A894)+1/4)              \  &A8DC
  &A89D    \         Directory-Track        /  &A8DD
  &A89E    /                                \  &A8DE
  &A89F               Format-Nr.               &A8DF
  &A8A0        Anzahl d. Sektoren pro Track    &A8E0
  &A8A1         GAP-Länge bei R/W             &A8E1
  &A8A2      GAP-Länge bei Formatierung       &A8E2
  &A8A3            Format-Füllbyte            &A8E3
  &A8A4             Sektorgrösse              &A8E4
  &A8A5            Records pro Sektor          &A8E5
  &A8A6       Aktueller Track d. Lesekopfs     &A8E6
  &A8A7            Head-Align Flag             &A8E7
  &A8A8      Flag f. Disk-Login (&FF=aus)      &A8E8
  &A8A9 - &A8B8        ???                     &A8E9 - &A8F8
  &A8B9 - &A8CF        ???                     &A8F9 - &A90F
```

1. Laufwerkstabelle

```
  &A910 - &A917        ???                     &A920 - &A927
  &A918    \      Zeiger auf Puffer1 (&A930)/  &A928
  &A919    /                                \  &A929
  &A91A    \      Zeiger auf 2. LW-Tabelle  /  &A92A
  &A91B    /                                \  &A92B
  &A91C    \      Zeiger auf &A8A9 / &A8E9  /  &A92C
  &A91D    /                                \  &A92D
  &A91E    \      Zeiger auf &A8B9 / &A8F9  /  &A92E
  &A91F    /                                \  &A92F
```

```
&A930 - &A9AF Puffer
&A9B0 - &AB7F Puffer
```

BASIC :

&AB80 - &ABFF CHR$(240) - CHR$(255)
&AC00 Flag f. zusaetzliche Blanks ignorieren
&AC01 READY-Patch
&AC04 Error-Patch
&AC07 Patch f. Befehl ausführen
&AC0A Patch f. Funktionsberechnung
&AC0D Patch f. Konstante holen (unbenutzt)
&AC10 Patch f. Eingabe, Zeile in Tokens wandeln
&AC13 Patch f. Ausgabe, Tokens listen
&AC16 Patch f. Eingabe, Umwandlung v. Ziffern
&AC19 Patch f. Operatoren
&AC1C Flag f. AUTO-Modus

&AC1F Step f. AUTO-Modus
&AC21 aktive Stream-Nr.
&AC22 Eingabe-Kanal
&AC23 Drucker-Posision
&AC24 WIDTH-Einstellung
&AC25 lfd. Position auf Datasette
&AC26 Flag f. ersten FOR-NEXT-Durchlauf
&AC27 - &AC2B Zwischenspeicher für FOR-Variablen
&AC2C Adresse des zugehörigen NEXT-Befehls
&AC2E Adresse des zugehörigen WEND-Befehls
&AC30 WHILE/WEND-Flag (&41 = WEND nicht benutzt, &04 = benutzt)
&AC32
&AC34 ON BREAK-Adresse
&AC36 Adresse f. KB-Eventblock
&AC38 - &AC43 Sound-Queue 0
&AC44 - &AC4F Sound-Queue 1
&AC50 - &AF5B Sound-Queue 2
&AC5C - &AC6D Eventblock 0
&AC6E - &AC7F Eventblock 1
&AC80 - &AC91 Eventblock 2
&AC92 - &ACA3 Eventblock 3
&ACA4 - &ADA3 Eingabepuffer (ohne Tokens)
&ADA4 ??
&ADA6 Adresse d. Fehler-Zeile
&ADA8 Programmzeiger nach Fehler
&ADAA Fehler-Nummer
&ADAB Programmzeiger nach Unterbrechung
&ADAD Zeilenadresse nach Unterbrechung
&ADAF Adresse der ON ERROR-Routine
&ADB1 Fehlerbehandlungs-Routine aktiv
&ADB2 SOUND-Parameter Kanal
&ADB3 SOUND-Parameter ENV-Kurve
&ADB4 SOUND-Parameter ENT-Kurve
&ADB5 SOUND-Parameter Periode
&ADB7 SOUND-Parameter Rauschen
&ADB8 SOUND-Parameter Lautstärke
&ADB9 SOUND-Parameter Dauer
&ADBA - &ADBB ENT+ENV
&ADCB - &ADCF Zwischenspeicher f. FLO-Zahl
&AED0 - &AE03 Tabelle f. skalare Variablen
&AE04 Zeiger FN-Tabelle
&AE06 - &AE0B Array-Tabelle
&AE0C - &AE25
&AE26 - &AE2D ??
&AE2D Trennzeichen bei INPUT

&AE2E Adresse bei READ
&AE30 DATA-Zeiger
&AE32 Speicher f. BASIC-Stack
&AE34 Adresse des lfd. Statements
&AE36 Adresse der lfd. Programmzeile
&AE38 TRACE-Flag (TRON/TROFF)
&AE3F Startadresse bei LOAD
&AE41 Flag f. CHAIN MERGE
&AE42 Filetyp
&AE43 Filelaenge
&AE45 Flag f. geschütztes BASIC-Programm
&AE46 - &AE71 Puffer f. Umwandlung ASCII
&AE72 Adresse f. CALL
&AE74 Konfiguration f. CALL
&AE75 HL waehrend CALL / Position im BASIC-Prg.
&AE77 SP waehrend CALL
&AE79 Tabulatorweite (ZONE)
&AE7B Zeiger auf HIMEM
&AE7D Zeiger auf Endadresse v. freien RAM
&AE7F Zeiger auf Startadresse v. freien RAM
&AE81 Zeiger auf BASIC-Programm Start
&AE83 Zeiger auf BASIC-Programm Ende
&AE85 Zeiger auf Variablenstart
&AE87 Zeiger auf Array-Start
&AE89 Zeiger auf Array-Ende
&AE8B - &B08A BASIC Stack
&B08B BASIC Stackpointer
&B08D Zeiger auf String-Start
&B08F Zeiger auf String-Ende
&B09A Stringdescriptor Stackpointer
&B09C - &B0B9 Stringdescriprot Stack
&B0BA - &B0C0 Stringdescriptor
&B0C1 Variablentyp
&B0C2 INT-Var. / Adr. v. FLO-Var. / Pointer v. STRING-Descriptor
&B0C4 - &B0FF ??

Kernel :

&B100 Start Int. Pending Queue
&B104 diverse Flags f. interne Routinen
&B105 SP
&B107 Temporärer Stack
&B187 Timer low
&B189 Timer hi
&B18B Timer Flag
&B18C Start Framefly Chain
&B18E Start Fast Ticker Chain
&B190 Start Ticker Chain
&B192 Ticker-Count
&B193 Start syncrone Pending-Queue
&B195 Priorität d. lfd. Events
&B196 Auszuführender Befehl (ROM-Konfig-Sicherung)
&B197 - &B1A7 ??
&B1A8 lfd. Expansions-ROM
&B1A9 lfd. ROM-Einsprung
&B1AB lfd. ROM-Konfig.
&B1AC ROM1-Eintrag IY-Wert (z.B. Adress-Tabelle)
&B1AE ROM2 IY
&B1B0 ROM3 IY
&B1B2 ROM4 IY
&B1B4 ROM5 IY

```
&B1B6 ROM6 IY
&B1B8 ROM7 IY
&B1BA - &B1C7 frei
```

Screen :

```
&B1C8 Screen Mode
&B1C9 Screen Scroll Offset
&B1CA Screen Startadresse
&B1CC Write indirection
&B1CF Bitmasken f. Screen (abh. von Mode)
&B1D7 Blinkperiode 2. Farbe  \
&B1D8 Blinkperiode 1. Farbe   \
&B1D9 Farbspeicher 2. Farben  / siehe Kapitel 1.1
&B1EA Farbspeicher 1. Farben /
&B1FB Flag f. lfd. Farbsatz
&B1FC Flag
&B1FD ??
&B1FE Event Block f. Inks
```

Text :

```
&B20C lfd. Bildschirmfenster
&B20D Parameter Fenster 0
&B21C Parameter Fenster 1
&B22B Parameter Fenster 2
&B23A Parameter Fenster 3
&B249 Parameter Fenster 4
&B258 Parameter Fenster 5
&B267 Parameter Fenster 6
&B276 Parameter Fenster 7
&B285 Cursorposition (y,x)
&B287 Fenster-Flag (0 = Bildschirm)
&B288 aktuelles Fenster oben
&B289 aktuelles Fenster links
&B28A aktuelles Fenster unten
&B28B aktuelles Fenster rechts
&B28C lfd. Roll Count
&B28D lfd. Cursor Flag
&B28E VDU-Flag (0 = disabled)
&B28F aktueller PEN-Wert
&B290 aktueller PAPER-Wert
&B291 lfd. Background-Mode
&B293 Grafikzeichen-Schreibmode (0 = disabled)
&B294 1. Zeichen d. Usermatrix ( (&B295)=0 -> norm )
&B296 Adr. d. Usermatrix
&B298 ??
&B2B8 Zeichenzähler Ctrl-Puffer
&B2B9 Start Ctrl-Puffer
&B2C8 Sprungtabelle Steuerzeichen
```

Graphics :

```
&B328 Origin-Wert X
&B32A Origin-Wert Y
&B32C aktuelle X-Koordinate
&B32E aktuelle Y-Koordinate
&B330 X-Koordinate f. Grafikfenster rechts
```

&B332 X-Koordinate f. Grafikfenster links
&B334 Y-Koordinate f. Grafikfenster oben
&B336 Y-Koordinate f. Grafikfenster unten
&B338 Grafik-Pen
&B339 Grafik-Paper
&B342 Rechenpuffer X-Koordinate
&B344 Rechenpuffer Y-Koordinate

Key Manager :

&B4DC Ende der KM-Tabellen
&B4DE Exp.-String Pointer
&B4E0 Put Back Puffer
&B4E1 Start d. Expansionspuffers
&B4E3 Ende d. Expansionspuffers
&B4E5 Start f. freier Expansionspuffer
&B4E7 Status Shift-Lock
&B4E8 Status Caps-Lock
&B4E9 Tasten-Delay
&B4EB Key State Map
&B4ED Key 16 bis 23
&B4F1 JOY1-Status
&B4F4 JOY0-Status
&B4F5 während Scan gedrückte Tasten
&B4FF Multihit Control zu &B4F5
&B50D Break Event-Block
&B541 Adr. Translations-Tabelle
&B543 Adr. SHIFT-Tabelle
&B545 Adr. CTRL-Tabelle
&B547 Adr. Repeat-Tabelle

Sound :

&B551 alte Sound-Aktivität (nach HOLD)
&B552 lfd. Sound-Aktivität
&B555 Sound-Eventblock
&B55C Parameter Kanal A
&B59B Parameter Kanal B
&B5DA Parameter Kanal C
&B60A ENV-Kurven
&B6FA ENT-Kurven
&B6FB - &B6FF ??
&B700 - &B7FF Soundpuffer

Cassette :

&B800 Message Flag
&B802 Input-Puffer Status
&B803 Startadr. d. Input-Puffers
&B805 Pointer f. Input-Puffer
&B807 File Header Input
&B847 Output-Puffer Status
&B848 Startadr. d. Output-Puffers
&B84A Pointer f. Output-Puffer
&B84C File Header Output
&B8D1 Speed

&B8DD EDIT Insert Flag

&B8E3 - &B8DE frei ?
&B8E4 - &B8E7 RND-Wert
&B8E8 - &B8EC Zwischenspeicher f. FLO-Zahl
&B8ED - &B8F1 Zwischenspeicher f. FLO-Zahl
&B8F2 - &B8F6 Zwischenspeicher f. FLO-Zahl
&B8F7   Flag f. DEG/RAD - Status
&B8F8 - &B8FF

Kernel :

&B900 Jump zu Upper-ROM Enable (BASIC-ROM)
&B903 Jump zu Upper-ROM Disable
&B906 Jump zu Lower-ROM Enable (Betr.sys.)
&B909 Jump zu Lower-ROM Disable
&B90C Jump zu ROM Restore
&B90F Jump zu ROM Select (Enable Exp.ROM) C=Romnr.
&B912 Jump zu Current Selection (Ausgangswert in A)
&B915 Jump zu Probe ROM
&B918 Jump zu ROM Deselect
&B91B Jump zu LDIR-Routine
&B91E Jump zu LDDR-Routine
&B921 Poll Sync.
&B939 RST7-Routine Interrupt Entry
&B970 Ext. Interrupt Entry
&B97C Low Pchl.Far Routine
&B982 RST1-Routine Low Jump
&B9A8 Far Call Continued
&B9B1 Pchl.Far Routine
&B9B9 Far ICall Routine
&B9BF RST3-Routine Far Call
&BA10 Side Call Continued
&BA16 RST2-Routine Side Call
&BA2E RST5 Firm Jump Routine
&BA4A Lower-ROM Enable
&BA54 Lower-ROM Disable
&BA5E Upper-ROM Enable
&BA68 Upper-ROM Disable
&BA72 ROM Restore
&BA7E ROM Select
&BA83 Probe ROM
&BA8C ROM Deselect
&BAA2 Current Selection
&BAA6 LDIR-Routine
&BAAC LDDR-Routine
&BAB2 ROM off + Config.Save
&BAC7 CALL (HL)
&BACB RST4 RAM LAM (LD a,(hl))
&BADC RAM LAM (IX) (ld a,(ix))
&BAE9 - &BAFF frei

5.7.2. Die Firmware

KM (Key Manager) :

```
&BB00 KM Initialize
&BB03 KM Reset (Zeichenpuffer löschen)
&BB06 KM Wait Char (Auf Zeichen warten -> O: A=CHR$-Nr.)
&BB09 KM Read Char Liest Zeichen, falls vorhanden (O: A=CHR$-Nr.)
&BB0C KM Char Return Hinterlegt Zeichen im Puffer
&BB0F KM Set Expand
&BB12 KM Get Expand
&BB15 KM Exp. Puffer
&BB18 KM Wait key  (Effektiv wie &BB06)
&BB1B KM Read key  (Wie &BB09, aber O: A=Tastennr.)
&BB1E KM Test key  (I: A=Tastennr., O: A=0 nicht gedrückt; A=1 gedrückt)
&BB21 KM Get Shift state
&BB24 KM Get Joystick  (O: HL=Joystick 0, DE=Joystick 1)
&BB27 KM Set translate (Tastatur 1. Ebene)
&BB2A KM Get translate (Tastatur 1. Ebene)
&BB2D KM Set Shift (2. Ebene)
&BB30 KM Get Shift (2. Ebene)
&BB33 KM Set Ctrl (3. Ebene)
&BB36 KM Get Ctrl (3. Ebene)
&BB39 KM Set Repeat (f. bestimmte Taste)
&BB3C KM Get Repeat (f. bestimmte Taste)
&BB3F KM Set delay
&BB42 KM Get delay
&BB45 KM Arm Break
&BB48 KM Disarm Break
&BB4B KM Break Event
```

TXT (Text) :

```
&BB4E TXT Initialize   Zeichensatz zurücksetzen
&BB51 TXT Reset
&BB54 TXT VDU Enable
&BB57 TXT VDU Disable
&BB5A TXT Output (Zeichen in A darstellen oder ausführen)
&BB5D TXT Write char (darstellen)
&BB60 TXT Read char
&BB63 TXT Set graphic (Steuerzeichen an/aus)
&BB66 TXT Window enable (Aktuelle Fenstergrösse definieren)
&BB69 TXT Get window
&BB6C TXT Clear window
&BB6F TXT Set column
&BB72 TXT Set row
&BB75 TXT Set cursor (I: L=Zeile, H=Spalte)
&BB78 TXT Get cursor (O: L=Zeile, H=Spalte)
&BB7B TXT Cursor enable  \ Anwenderprg.
&BB7E TXT Cursor disable /
&BB81 TXT Cursor on      \ Betriebssystem
&BB84 TXT Cursor off     /
&BB87 TXT Validate
&BB8A TXT Place cursor   Cursor setzen
&BB8D TXT Remove cursor  Cursor löschen
&BB90 TXT Set pen    (I: A=Stiftnr.)
&BB93 TXT Get pen    (O: A=Stiftnr.)
&BB96 TXT Set paper  (I: A=Stiftnr.)
&BB99 TXT Get paper  (O: A=Stiftnr.)
```

```
&BB9C TXT Inverse
&BB9F TXT Set background \ Transparentmodus
&BBA2 TXT Get background /
&BBA5 TXT Get Matrix
&BBA8 TXT Set Matrix (I: HL=Adresse der 8 Bytes, A=CHR$-Nr.)
&BBAB TXT Set Matrix Table (entspricht der 'SYMBOL AFTER'-Anweisung)
&BBAE TXT Get Matrix Table
&BBB1 TXT Get Controls (Adr. f. Control-Zeichen-Sprungtabelle)
&BBB4 TXT Str Select (Textfenster wählen)
&BBB7 TXT Swap streams (Fenster tauschen)
```

GRA (Graphics) :

```
&BBBA GRA initialize
&BBBD GRA Reset
&BBC0 GRA Move absolute  (I: DE=X, HL=Y)
&BBC3 GRA Move relative        .
&BBC6 GRA Ask cursor           .
&BBC9 GRA Set origin           .
&BBCC GRA Get origin           .
&BBCF GRA Set win width
&BBD2 GRA Set win height
&BBD5 GRA Get win width
&BBD8 GRA Get win height
&BBDB GRA Clr win
&BBDE GRA Set pen        (I: A=Stiftnr.)
&BBE1 GRA Get pen              .
&BBE4 GRA Set paper            .
&BBE7 GRA Get paper            .
&BBEA GRA Plot absolute  (Parameter wie bei &BBC0)
&BBED GRA Plot relative        .
&BBF0 GRA Test absolute        .
&BBF3 GRA Test relative        .
&BBF6 GRA Line absolute        .
&BBF9 GRA Line relative        .
&BBFC GRA Write char     (I: A=CHR$-Nr.)
```

SCR (Screen) :

```
&BBFF SCR Initialize
&BC02 SCR Reset         Inks zurücksetzen
&BC05 SCR Set offset
&BC08 SCR Set base      I: A=Highbyte d. Screenadr.
&BC0B SCR Get location
&BC0E SCR Set mode      I: A=Modenr.
&BC11 SCR Get mode      O: A=Modenr.
&BC14 SCR Clear
&BC17 SCR Char limits (max. Zeilen- u. Spaltenzahl)
&BC1A SCR Char pos.     I: H=Spalte, L=Zeile ; O: HL=Bildschirmadr.
&BC1D SCR Dot pos.      I: DE=X, HL=Y ; O: HL=Bildschirmadresse
&BC20 SCR Next byte
&BC23 SCR Previous byte
&BC26 SCR Next line
&BC29 SCR Previous line
&BC2C SCR Ink encode
&BC2F SCR Ink decode
&BC32 SCR Set ink       I: A=Stiftnr., B,C = Farbnr.
&BC35 SCR Get ink       O:          .
&BC38 SCR Set border    I:          .
```

```
&BC3B SCR Get border    O:          .
&BC3E SCR Set flashing
&BC41 SCR Get flashing
&BC44 SCR Fill box (Zeichenbezogen)
&BC47 SCR Flood box (Bildschirmadr.bezogen)
&BC4A SCR Char invert
&BC4D SCR Hardware scroll
&BC50 SCR Software scroll
&BC53 SCR Unpack \ f. Mode 1/0
&BC56 SCR Repack /
&BC59 SCR Access (Control-Z. sichtbar/unsichtbar)
&BC5C SCR Pixels
&BC5F SCR Horizontal \ Linie
&BC62 SCR Vertical   /
```

CAS (Casette) :

```
&BC65 CAS Initialize
&BC68 CAS Set speed    I: A, HL (s. Kapitel 3.4)
&BC6B CAS Noisy (Discleseversuche / Datasettenmeldungen an/aus)
&BC6E CAS Start motor
&BC71 CAS Stop motor
&BC74 CAS Restore motor
&BC77 CAS In open      I: HL = Name, DE=Ladeadr. B = Namenslänge)
&BC7A CAS In close
&BC7D CAS In Abandon
&BC80 CAS In Char (Puffer)
&BC83 CAS In direct    I: HL = Adresse, O: HL = Autostartadresse)
&BC86 CAS Return (zuletzt gelesenes Zeichen zurück in Puffer)
&BC89 CAS Test EOF (End Of File)
&BC8C CAS Out open (Parameter wie bei &BC77)
&BC8F CAS Out close
&BC92 CAS Out abandon
&BC95 CAS Out char (Puffer)
&BC98 CAS Out direct   I: HL=Startadr, DE=Länge, BC=Autostartadr., A=Filetyp
&BC9B CAS Catalog      I: HL=Pufferadr.
&BC9E CAS Write        I: HL=Startadr., DE=Länge, A=Syncronbyte
&BCA1 CAS Read         I: HL=Startadr., DE=Länge, A=Syncronbyte
&BCA4 CAS Check
```

Sound :

```
&BCA7 SND Reset (Alle Töne abschalten)
&BCAA SND Queue (Ton an Queue anhängen)
&BCAD SND Check
&BCB0 SND Arm Event
&BCB3 SND Release
&BCB6 SND Hold
&BCB9 SND Continue
&BCBC SND Amplitude Envelope (ENV)
&BCBF SND Tone Envelope (ENT)
&BCC2 SND ENV adress (Adresse f. Kurve holen)
&BCC5 SND ENT adress (Adresse f. Kurve holen)
```

KL (Kernel) :

```
&BCC8 KL Choke off (reset)
&BCCB KL ROM Walk
```

```
&BCCE KL Init back
&BCD1 KL Log ext. (s. Kapitel 4.2)
&BCD4 KL Find command  I: A=Tokennr.  O: HL=Adresse
&BCD7 KL New framefly                          \
&BCDA KL Add framefly                          !
&BCDD KL Del framefly                          !
&BCE0 KL New fast ticker                       !
&BCE3 KL Add fast ticker                       !
&BCE6 KL Del fast ticker                       !
&BCE9 KL Add ticker                            !
&BCEC KL del ticker                            !
&BCEF KL Init event                             \
&BCF2 KL Event (kicken)                          \  s. Kapitel 4.3
&BCF5 KL Sync reset (Pending Queue löschen)      /
&BCF8 KL Del sync                               /
&BCFB KL Next sync                             !
&BCFE KL Do sync                               !
&BD01 KL Done sync                             !
&BD04 KL Event disable                         !
&BD07 KL Event enable                          !
&BD0A KL Disarm event                          /
&BD0D KL Time please (4Byte-Wert in HL+DE)
&BD10 KL Time set    (4Byte-Wert in HL+DE)
```

MC (Machine) :

```
&BD13 MC Boot program
&BD16 MC Start program
&BD19 MC Wait flyback (Strahlenrücklauf abwarten)
&BD1C MC Set mode (A=Modenr.)
&BD1F MC Screen offset
&BD22 MC Clear inks
&BD25 MC Set inks
&BD28 MC Reset printer
&BD2B MC Print char (Drucker)
&BD2E MC Busy printer
&BD31 MC Send printer (Queue)
&BD34 MC Sound Register
&BD37 MC Jump restore
```

5.7.3. Basic-Vektoren

FLO (Flowing Point) :

```
&BD3A FLO EDIT
&BD3D FLO HL:=DE
&BD40 FLO Int ->FLO
&BD43 FLO 4-Byte-Wert ->FLO
&BD46 FLO FLO->Int
&BD49 FLO Flo->Int
&BD4C FLO FIX
&BD4F FLO INT
&BD52 FLO ??
&BD55 FLO *10^x (mal  10 hoch x
&BD58 FLO Addition                 \
&BD5B FLO Subtraktion              ! HL:=HL ? DE
&BD5E FLO Subtraktion              !
&BD61 FLO Multiplikation           !
&BD64 FLO Division                 /
```

```
&BD67 FLO *2^x (mal  2 hoch x)
&BD6A FLO Vergleich
&BD6D FLO Vorzeichenwechsel
&BD70 FLO SGN
&BD73 FLO DEG/RAD
&BD76 FLO PI
&BD79 FLO Wurzel
&BD7C FLO Potenz
&BD7F FLO LOG
&BD82 FLO LOG10
&BD85 FLO EXP
&BD88 FLO SIN
&BD8B FLO COS
&BD8E FLO TAN
&BD91 FLO ATN
&BD94 FLO 4-Byte-Wert * 256 ->FLO
&BD97 FLO RND Init
&BD9A FLO Set RND seed
&BD9D FLO RND
&BDA0 FLO Get last RND value
```

INT (Intern) :

```
&BDA3 INT ??
&BDA6 INT ??
&BDA9 INT Vorz. in b
&BDAC INT Addition
&BDAF INT Subtraktion
&BDB2 INT Subtraktion
&BDB5 INT Multiplikation mit Vorzeichen
&BDB8 INT Division mit Vorzeichen
&BDBB INT MOD
&BDBE INT Multiplikation ohne Vorzeichen
&BDC1 INT Division ohne Vorzeichen
&BDC4 INT Vergleich
&BDC7 INT Vorzeichenwechsel
&BDCA INT SGN
```

Indirections :

```
&BDCD TXT Draw cursor
&BDD0 TXT Undraw cursor
&BDD3 TXT Write char (Zeichen abbilden)
&BDD6 TXT Unwrite (Zeichen lesen)
&BDD9 TXT Out action (Zeichen abbilden/ausführen)
&BDDC GRA Plot
&BDDF GRA Test
&BDE2 GRA Line
&BDE5 SCR Read
&BDE8 SCR Mode clear (CLS)
&BDEB ??
&BDEE KM  Test break (Break-Taste gedrückt?)
&BDF1 MC  Wait printer (Zeichen zum Drucker)
&BDF4 - &BE3F frei
```

DDI1 :

```
&BE40 R/W    Zeiger auf 1. LW-Tabelle
&BE42 R/W    Zeiger auf Tabelle v. letztem angemeldeten Laufwerk (&A890)
&BE44 R/W    Motor-Vorlaufzeit (Default &0032)
&BE46 R/W    Motor-Nachlaufzeit (Default &00FA)
&BE48 R/W    Schreib-Nachlauf (Default &AF -> 10 Microsec.)
&BE49 R/W    Kopf-Plazier-Zeit (Default &0F -> 1 Milisec.)
&BE4A R/W    Schrittgeschwindigkeit (Default &0C, max. &0A)
&BE4C R/W    Bit0=non-DMA Mode Bit1-Bit7=Kopf-Leseverzögerung (Default=&03)
&BE4D R/-    Disketten-Fehlernummer (siehe Kapitel 3.7)
&BE4E R/W    ??
&BE4F R/-    Letzter benutzter Block
&BE50 R/-    Kopf-Nummer (&00)
&BE51 R/-    Letzter benutzter Sektor
&BE52 R/-    Sektorgrösse
&BE53 R/W    Flag f. ??
&BE54 R/W    Flag f. ??
&BE55 R/W    Flag f. ??
&BE56 - &BE58 ??
&BE59 R/W    Flag f. ??
&BE5A R/-    ??
&BE5B R/-    ??
&BE5C R/-    ??
&BE5D        ??
&BE5E        ??
&BE5F R/W    Flag des Event-Blocks f. Motor drehen (&00 = aus)
&BE60 R/-    Zeiger Puffer1 ( (&BE7D)+&0230  ) (f. Directory-Einträge)
&BE62 R/-    Zeiger Puffer2 ( (&BE7D)+&02B0 Sector Puffer )
&BE64 R/W    Adresse f. Floppy Stack (Beginn)
&BE66 R/W    Anzahl der Leseversuche
&BE67        Interrupt-Eventblock (Ticker) f. Motor
&BE69 R/W    lfd. Zähler f. Vor-/Nachlauf (TickCount)
&BE6B        TickCount Reload
&BE6D        Pending Queue
&BE6F        Zähler
&BE70        Klasse (&80)
&BE71        Startadr. d. Routine (&D6C9, Floppy-ROM) + ROM-Konfig.
&BE74 R/-    Letzte benutzte Sektor-Nr.
&BE75 R/W    ??
&BE76 R/W    Adresse des Sector Puffers (Beginn)
&BE78 R/W    Flag f. Fehlerabfrage (Retry, Ignore or Cancel) (&FF=aus)
&BE79 - &BE7C ??
&BE7D R/W    Adresse des Floppy-Speichers (Beginn) (Default : &A700)
&BE7F - &BE81 Disc-Login Patch


&BE82 - &BFFF Stack (Stack ab &BFFF abwärts)
&C000 - &FFD0 Bildschirmspeicher
&FFD0 - &FFFF frei
```

5.8. Hardware-Daten

5.8.1. Joystick-Port

```
 01  02  03  04  05        01 oben     05 frei
                           02 unten    06 Feuer2
  *   *   *   *   *        03 links    07 Feuer1
                           04 rechts   08 COM1 (Joystick 0)
    *   *   *   *                      09 COM2 (Joystick 1)

    06  07  08  09       Mit OUT &F700,&82:OUT &7600,49 liegen zwischen
                         PIN 01-07 und PIN 08 für 1/50 sec 5 Volt an.
```

5.8.2. Druckerport (&EF d. 8255)

```
17 16 15 14 13 12 11 10 09 08 07 06 05  04 03 02 01

 *  *  *  *  *  *  *  *  *  *  *  *  *   *  *  *  *
 -------------------------------------  -----------
 *  *  *  *  *  *  *  *  *  *  *  *  *   *  *  *  *

35 34 33 32 31 30 29 28 27 26 25 24 23  22 21 20 19
```

```
01 Stobe (negiert)   11 BUSY
02 D0                12 NC
03 D1                13 NC
04 D2                14 GND
05 D3                15 NC
06 D4                16 GND
07 D5             19-33 NC
08 D6
09 D7
10 NC
```

5.8.3. Bildschirm-Ausgang

```
5 *         * 1       1 : Rot   - Signal
                      2 : Grün - Signal
                      3 : Blau  - Signal
       * 6            4 : Sync
                      5 : GND
4 *         * 2       6 : Ausleuchtung


     3 *
```

5.8.4. Expansions-Port

```
 49 47 45 43 41 39 37 35 33 31 29 27 25 23  21 19 17 15 13 11 09 07 05 03 01

  *  *  *  *  *  *  *  *  *  *  *  *  *  *   *  *  *  *  *  *  *  *  *  *  *
 ---------------------------------------------------------------------------
  *  *  *  *  *  *  *  *  *  *  *  *  *  *   *  *  *  *  *  *  *  *  *  *  *

 50 48 46 44 42 40 38 36 34 32 30 28 26 24  22 20 18 16 14 12 10 08 06 04 02
```

```
   01  Sound (mono)   17  A01      33* WR
   02  GND            18  A00      34* HALT
   03  A15            19  D07      35* INT
   04  A14            20  D06      36* NMI
   05  A13            21  D05      37* BUSRD
   06  A12            22  D04      38* BUSAK
   07  A11            23  D03      39  READY
   08  A10            24  D02      40* BUS RESET      49  GND
   09  A09            25  D01      41* RESET          50  TAKT 4MHz
   10  A08            26  D00      42* ROM EN
   11  A07            27  +5V      43  ROM DIS         * = negiert
   12  A06            28* MREQ     44  RAM RD
   13  A05            29* M1       45  RAM DIS
   14  A04            30* RFRSH    46  CURSOR
   15  A03            31* IORQ     47  L.PEN
   16  A02            32* RD       48* EXP
```

```
Adressbuss : A00-A15
  Datenbus : D00-D07
 Steuerbus :M1         (Machine CycleOne) zeigt an, dass Prozessor OP-Code vom
                       Datenbus liest.
            MREQ       (Memory Request) zeigt einen Schreib/Lesezugriff auf
                       den Speicher an.
            IORQ       zeigt einen Lese/Schreibzugriff auf den Port an
            RD         zeigt an, dass Prozessor lesen will
            WR         zeigt an, dass Prozessor schreiben will
            RESET      Reset aller externen Erweiterungen
            NMI        (NonMaskerable Interrupt) Hi-Lo-Impuls unterbricht
                       Programm (PC=&0066)
            BUSRQ      wenn Low --> nach Abarbeitung des letzten Befehls werden
                       A-Bus und D-Bus hochohmig. BUSAK wird Low. Hier könnte
                       2. Prozessor übernehmen. Hauptsächliche Nutzung f. DMA
            BUSAK      (Bus Acknowledge) wenn Low --> Zugriff kann erfolgen
            HALT       Nach OP-Code &76 low, wartet auf nächsten Interrupt
            RFRSH      zeigt an, dass auf dem A-Bus eine gültige Adresse liegt
            GND        Masse
            BUS RESET  wenn Low --> System Reset
            ROM EN     wenn Low --> Zugriff auf internes 32K-ROM
            ROM DIS    wenn Hi  --> kein Zugriff auf ROM
            RAM RD     wenn Low --> Zugriff auf RAM
            RAM DIS    wenn Hi  --> kein Zugriff auf RAM
            CURSOR     CRTC-Signal \
            L.PEN      CRTC-Signal / s. Kapitel 1.5
            EXP        Port B, Bit 4 des 8255
            INT        (IRQ) Interrupt Request, wenn Low -->Unterbricht
                       Programm (s. IM 0, Kapitel 5.6)
```