Für CPC 464/664/6128

SYBEX

Das Schneider CPC Systembuch



Günter Woigk

Das Schneider CPC Systembuch

Das Schneider CPC Systembuch

Günter Woigk



DÜSSELDORF · SAN FRANCISCO · PARIS · LONDON

Satz: SYBEX-Verlag GmbH, Düsseldorf
Umschlaggestaltung: tgr – typo-grafik-repro gmbh, Remscheid
Gesamtherstellung: Boss-Druck und Verlag, Kleve

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen zu publizieren. SYBEX-Verlag GmbH, Düsseldorf, übernimmt keine Verantwortung für die Nutzung dieser Informationen, auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren.

ISBN-Nr. 3-88745-606-8
1. Auflage 1987

Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Printed in Germany
Copyright © 1987 by SYBEX-Verlag GmbH, Düsseldorf

Inhaltsverzeichnis

Einleitung 15

Die Grafik 15
Die Farben 17
Die Tastatur 18
Der Sound 19
Die Massenspeicher 21
Der Speicher 23
BASIC 24
LOGO 27
CP/M 29

Kapitel 1: Grundlagen 33

Datenspeicherung und Datenstrukturen 33
Records 34
Arrays 35
Arrays aus Vl-Records 36
Chains 37
Trees 40
Stacks: Last In – First Out 42
Queues: First In – First Out 46

Datentypen 53
BCD-Codierung 53
Packed BCD 54
Bytes 54
Words-Integer 55
Real 56
Die Mantisse 57
Der Exponent 57
Null 58
Dezimal Wandlung 59
Strings 60
Garbage Collection 61

Programmiertechnik 65
Pflichtenliste 65
Gliederung 65
Programmbibliotheken 65
Modularisierung 65
Kommentare 65

Programmstrukturen 66
Verzweigungen – der einfache Sprung 67
Bedingte Bearbeitung von Befehlen 68
Indirekter Sprung – Verzweigung via Tabelle 70
Iterationen – Schleifen 72

Unterprogramme 76
Variablen 79
Übergabe von Argumenten und Ergebnissen 81
Rekursion – Selbstaufruf einer Prozedur 84
Interrupts – Unterbrechungen 87

Befehlselemente 89
Commands 90
Statements 91
Separatoren 93
Klammern 94
Funktionen, Argumente 94
Felder, Indizes 95
Operatoren, Operanden 95

Syntax und Semantik 98

Kapitel 2: Das Innenleben der CPC-Rechner 101

Die CPU Z80 101
Beschreibung des Z80 101
Die Anschlußbelegung der CPU Z80 104
Adressierungsarten des Z80 107
Datenbreite 109
Die verschiedenen Interrupt-Modi des Z80 110
Das Refresh-Register 111
Der zweite Registersatz 112
Unterprogrammaufrufe 113
16-Bit-Worte speichern 114
Besonderheiten des Z80 im Schneider CPC 115

Die PIO 8255 121
Die Anschlußbelegung der PIO 8255 121
Die drei verschiedenen Betriebsarten der PIO 8255 124
Anschluß der PIO 8255 an das Gesamtsystem 125

Der PSG AY-3-8912 131
Die Anschlußbelegung des AY-3-8912 131
Funktionsauswahl im PSG über BC1, BC2 und BDIR..... 134
Die Register des AY-3-8912 137

Der Video-Controller HD 6845 142
Die Anschlußbelegung des CRTC HD 6845 144
Port-Adressen des 6845 147
Normaler Einsatz des HD 6845 148
Der Einsatz des HD 6845 im Schneider CPC 150
Die Register des Video-Controllers HD 6845 152
Erklärungen zu den einzelnen Registern 154

Die ULA 40007, 40008 oder 400010 und das PAL HAL 16L8 158
Die Anschlußbelegung
der ULA 40007 und 40008 (CPC 464 und 664) 159
Die Anschlußbelegung der ULA 40010 (CPC6128) 160
Die Register des Gate Arrays und des PALs 164

Der FDC 765 170
Die Anschlußbelegung des FDC 765 172
Ansteuerung des Disketten-Controllers 177
Programmierung des FDC 765 179
Die Befehle des FDC 765 186
Organisation einer Spur durch den FDC 765 195
Eine Spur auf der Diskette..... 196
Ein Sektor in einer Spur 196
Das Identifikationsfeld in jedem Sektor 197
Die Besonderheiten des FDC 765 im Schneider CPC 199
FDC-Praxis 201

Der Schreib-/Lesespeicher – das RAM..... 203
Die Anschlußbelegung der RAMs 4164 203
Refresh der dynamischen RAMs 204
Adressierungsarten
der dynamischen RAM-ICs im Schneider CPC 207
Die Festwertspeicher – die ROMs 208
Anschlußbelegung der Eproms 27128 und 27256 210

Programmierung der Eproms 211
27256-Arbeitsmodi 213
27128-Arbeitsmodi 213
Timing-Diagramm "Programming 27128" 214
Timing-Diagramm "Programming 27256" 214

Die Schnittstellen der Schneider CPC-Rechner 215
Die Stromversorgung 215
Das Monitorsignal 215
Der Audio-Anschluß 216
Der Joystick-Anschluß 217
Der Drucker-Port 220
Der Expansion-Port (Systembuch) 224
Der Anschluß für den Kassettenrecorder 228
Der Anschluß für das zweite Diskettenlaufwerk 230

Kapitel 3: Das Betriebssystem des Schneider CPC 235

Die Speicherkonfiguration im Schneider CPC 235
Die ROM-Konfiguration 238
Externe Hardware 242

Die Aufteilung des RAMs im Schneider CPC 249
Die Speicherbelegung durch das Betriebssystem 249
Speicheraufteilung durch ein Vordergrund-Programm 252
Die Aufteilung des RAMs durch den BASIC-Interpreter 255

Der BASIC-Interpreter 260
Der Programmbereich 260
Der Variablenbereich 272
Verwaltung der Programmstrukturen 276

Maschinencode auf dem CPC 281
Vordergrund-Programme vom Massenspeicher 281
ROM-Software 282
Hintergrund-Routinen von Massenspeichern 285
Resident External Extensions 285

BASIC und Maschinencode 289
Maschinencode im String 289
Maschinencode über HIMEM 291
Parameter 293

Hilfreiche Fehlermeldungen 297
RSX-Loader 298
Z80-Relocalisator 300

Die Sprungleisten des Betriebssystems 307
Jumpblocks – Sprungleisten 309
Befehlserweiterung mit Restart 309
Patchen von Vektoren 312
Die Sprungleisten 317
High Kernel Jumpblock 320
Main Firmware Jumpblock 321

Die Abteilung des Betriebssystems 322
Der Key-Manager 323
Die Text-VDU 331
Die Grafik-VDU 340
Das Screen Pack 354
Der Kassetten-Manager 367
Der Sound-Manager 401
Der Kernel – Software-Interrupts 424
Die BASIC-Vektoren 445

Kapitel 4: Die Firmware des Schneider CPC 455

THE MAIN JUMPBLOCK – Der zentrale Sprungbereich 455
THE KEY MANAGER (KM) – Die Tastatur-Routinen 455
THE TEXT VDU (TXT) – Die Textausgabe-Routinen 457
THE GRAPHICS VDU (GRA) – Die Grafik-Routinen 460
THE SCREEN PACK(SCR) – Die Bildschirm-Routinen 462
THE CASSETTE MANAGER (CAS)
Die Kassetten- (und Disketten-) Routinen 465
THE SOUND MANAGER SOUND)
Die Lautsprecher-Routinen 467
THE KERNEL (KL) – Die Zentrale 468
THE MASCHINE PACK (MC) – Maschinennahe Routinen 470
JUMPER (JUMP) – Eine Routine für die Sprungleiste 471

Die Indirections der Firmware Packs 471

THE HIGH KERNEL JUMPBLOCK
Die obere Sprungleiste des Kernel 473

THE LOW KERNEL JUMPBLOCK
Die untere Sprungleiste des Kernel 474

Die BASIC-Vektoren 476
Der Editor 476
Die Fließkomma-Routinen 477
Die Integer-Routinen 478
Konvertierungs-Routinen 478
Zusätzliche Vektorens im Schneider CPC 464 und 6128 479
Zusätzliche Indirection im Schneider CPC 464 und 6128 480
Zusätzliche Vektoren im Schneider CPC 6128 480

THE KEY MANAGER (KM) – Die Tastatur-Routinen 480
THE TEXT VDU (TXT) – Die Textausgabe-Routinen 488
THE GRAPHICS VDU (GRA) – Die Grafik-Routinen 502
THE SCREEN PACK (SCR) – Die Bildschirm-Routinen 514
THE CASSETTE MANAGER (CAS)
Die Kassetten- (und Disketten-) Routinen 527
THE SOUND MANAGER (SOUND)
Die Lautsprecher-Routinen 537
THE KERNEL (KL) – Die Zentrale 542
THE MASCHINE PACK (MC) – Maschinennahe Sprungleiste 557
Jumper (Jump) – Eine Routine für die Sprungleiste 563

Die Indirections der Firmware Packs 563

THE HIGH KERNEL JUMPBLOCK
Die obere Sprungleiste des Kernel 569

THE LOW KERNEL JUMPBLOCK
Die untere Sprungleiste des Kernel 573

Die BASIC-Vektoren 582
Der Editor 582
Die Fließkomma-Routinen 583
Die Integer-Routinen 591
Konvertierungs-Routinen 593

Anhang A: Hardware-Basteleien 597

Das 8. Bit 597
Externes RAM 599
Bastelanleitung für RD- und WR- wirksames RAMDIS 600
Ein RESET-Knopf 601

Anhang B: Systemspeicher CPC 603
Das RAM des Betriebssystems 603
RAM des Kernel 603
RAM des Machine Pack 603
RAM des Screen Pack 604
RAM der Text-VDU 604
RAM der Graphics-VDU 605
RAM des Keyboard-Managers 605
RAM des Sound-Managers 606
RAM des Cassette-Managers 607
RAM des Zeileneditors 607
RAM des Floating Point Pack 607

Das RAM des BASIC-Interpreters 608

Das RAM des AMSDOS-Disketten-Controllers 610
Die ortsfesten Systemspeicher 610
Das dynamisch zugeteilte RAM 611

Anhang C: Der Z80 615

Die Befehle des Z80, sortiert nach ihrem Opcode 615
Befehlssatz des Z80, sortiert nach Funktionen 621
Wirkung der Z80-Befehle auf Flags 624

Befehlscodes der Z80-Befehle nach Opcode-Gruppen 625
Illegals 628

Ausführungszeiten für die einzelnen Befehle des Z 80 628
Die Register des Z80 631
Timing-Diagramme
für die Speicher- und Peripheriezugriffe des Z80 632

Anhang D: Speicher und Peripherie 637

Die Speicheraufteilung im CPC 637
Reservierte und frei verfügbare I/O-Adressen 638
Die PIO 8255 639
Die ULA 640
Das PAL im CPC 6128 641
Das Eprom 27256 641

Das Eprom 27128 642
Die Anschlüsse am Schneider CPC 642
Der Monitoranschluß 642
Der Audio-Anschluß 642
Der Joystick-Anschluß 643
Der Drucker-Port 643
Der Expansions-Port (Systembus) 644
Der Anschluß für den Kassettenrecorder am CPC 664 und 6128 .. 644
Der Recorderanschluß im CPC 464 645
Der Anschluß für das zweite Diskettenlaufwerk 645

Die Anschlußbelegungen der wichtigsten ICs im CPC 646
Die CPU Z80 646
Die Eproms 27128 und 27256 646
Die PIO 8255 647
Die ULA 40007 und 40008 (CPC 464 und 664) 647
Die ULA 40010(CPC 6128) 648
Das RAM 4164 648
Der CRTC HD 6845 649
Der PSG AY-3-8912 649
Der FDC 765 650

Anhang E: Die Tastatur 651

Tastennummern 651
Schaubilder der Tastatur 651

Die Standard-Tastenübersetzung 655
Erweiterungszeichen 657
Steuerzeichen des Key-Managers und des Zeileneditors 658

Anhang F: Die Bildausgabe 659

Die Controlcodes 659

Der CRTC HD 6845 662
Tinten und Farben 662

Standard-Zuordnung der Tinten zu Farbnummern 663
Zusammenhang zwischen Farbe,
Farbnummer und Paletten-Farbnummer 663

Die ULA 664
Die Codierung der Tintennummern in den Bildschirm-Bytes 664
Pixel-Masken 665
Farbmasken (encoded inks) 666

Der Zeichensatz des Schneider CPC 667

Anhang G: Die Tonausgabe 675

Ansteuerung des PSG 675
Das Kontrollregister (Register 7) 675
Die Register des AY- 3-8912 676
Die möglichen Hüllkurvenformen (Register 13) 676
Periodenlängen der Noten aus 9 Oktaven 677

Anhang H: Die Floppy 683

Port-Adressen zum Floppy-Controller 683

Die Register des FDC 683
Das Haupt-Statusregister 683
Das Statusregister 0 683
Das Statusregister 1 684
Das Statusregister 2 684
Das Statusregister 3 684

Programmierung des FDC 685
OUT Standard-Datenblock 685
INP Standard-Datenblock 685
Die Befehle des FDC 685

Anhang I: BASIC 687

Die Token des Locomotive-BASIC 687
Priorität der Operationen
in arithmetischen Ausdrücken in BASIC 691

Stichwortverzeichnis 693

Einleitung

Durch die drei verschiedenen Arten der Bildschirmdarstellung sind die CPCs wie nur wenige Homecomputer sowohl für Spiele und Grafiken als auch für Textverarbeitung geeignet. Im folgenden bringen wir einen kurzen Abriß über die grundlegenden Eigenschaften der Schneider CPCs.

Die Grafik

Modus 0 ist mit seinen 16 verschiedenen Farben und einer recht groben Auflösung, die nur 20 Zeichen pro Zeile zuläßt, hauptsächlich für Video-Spiele gedacht.

Viele Spiele begnügen sich aber auch mit weniger Farben, um dadurch in den Genuß der mittleren Auflösung zu kommen: Im Modus 1 sind vier Farben bei 40 Zeichen pro Zeile darstellbar. Für Grafiken ist diese Stufe meist der beste Kompromiß zwischen Auflösung und Farbenvielfalt. Dieser Modus wird auch automatisch eingestellt, wenn man den Computer einschaltet.

Texte wird man aber meist im Modus 2 bearbeiten. Der bietet zwar nur zwei verschiedene Farben, aber eine so hohe Auflösung, daß 80 Buchstaben in einer Zeile untergebracht werden können. Hat man einen Farbmonitor oder einen Fernseher angeschlossen, langt deren Auflösung aber meist nicht aus, die 80 Zeichen auch lesbar abzubilden. Dann muß man auch für die Textdarstellung auf den 40-Zeichen-Modus zurückgreifen. Unabhängig von der horizontalen Auflösung bleibt die Zeilenzahl immer gleich: In allen drei Modi stehen 25 Zeilen für die Darstellung von Text zur Verfügung.

Nun ist die Bildschirmdarstellung beim Schneider CPC aber nicht auf Text und Grafiksymbole beschränkt. Der Schneider CPC kennt eigentlich nur eine 100%ige Grafikdarstellung. Auch die Buchstaben werden sozusagen auf den Bildschirm gemalt.

Sehr viele Computer haben nur einen Textbildschirm, auf dem, wenn überhaupt, nur mit großem Aufwand echte Grafik dargestellt werden kann. Das liegt daran, daß in ihrem Bildschirmspeicher nur die Codes für die darzustellenden Zeichen gespeichert werden. Hat die erste Zelle im Video-RAM beispielsweise den Wert 65, so wird in der linken oberen Ecke des Monitorbildes ein großes A dargestellt.

Die eine Speicherzelle mit dem Wert 65 reicht aber nicht aus, um festzulegen, wie das große A aussehen soll. Die Information genügt gerade, um zu bestimmen, daß es ein großes A sein soll. Das Aussehen aller Buchstaben und Sonderzeichen muß in einem zusätzlichem Speicher, dem sogenannten *Character ROM* festgelegt werden.

Beim Schneider CPC ist das jedoch anders. Hier ist der Bildschirmspeicher viel größer und enthält direkt alle Informationen darüber, wie die Zeichen auf dem Monitor darzustellen sind. Je nach Modus sind 8, 16 oder sogar 32 Speicherzellen dafür zuständig, festzulegen, wie ein Buchstabe dargestellt wird. Daß dies ein bestimmtes Zeichen ist, interessiert die Bildschirmdarstellung überhaupt nicht. Deshalb ist es auch kein Problem, Text mit Grafik zu mischen, denn der Text wird ja als Grafik dargestellt.

Das hat natürlich nicht nur Vorteile, sondern leider auch eine ganze Menge Nachteile. So ist der Speicherbedarf für den Bildwiederholspeicher erheblich größer: Ein Textbildschirm benötigt für die Darstellung von 2000 Zeichen auch einen Bildwiederholspeicher mit 2000 Speicherzellen.

Der Schneider CPC braucht, um die Grafikinformation direkt bereitzuhalten, 16000 Bytes. Diese 16000 Bytes gehen dem Benutzerspeicher verloren. Unter CP/M hat man deshalb mit manchen Programmen Schwierigkeiten, die einfach für mehr frei verfügbaren Speicherplatz ausgelegt sind.

Außerdem wird die Textausgabe entschieden langsamer. Die CPU muß ja nicht nur den Zeichencode in den Bildschirmspeicher schreiben, sondern erst einmal selbst nachschauen, wie das Zeichen aussieht. Dazu sind im ROM des CPC Grafikmatrizen für die insgesamt 256 verschiedenen Zeichen gespeichert. Diese Grafikinformation muß, entsprechend dem eingestellten Modus und der gewählten Farben, mit komplizierten Operationen in die endgültige Grafikinformation für den Bildschirm umgewandelt werden.

Der Aufwand, ein Zeichen darzustellen, ist beim Schneider CPC um Größenordnungen höher als bei einem reinen Textsystem. Dementsprechend langsam ist auch die Zeichenausgabe auf dem Bildschirm.

Entsprechend umfassend sind allerdings die Grafikfähigkeiten der CPCs.

Die Auflösung in Y-Richtung (senkrecht) ist wie bei der Textdarstellung in allen drei Modi gleich: Es gibt insgesamt 200 verschiedene Grafikzeilen. Umgerechnet ergibt sich, daß ein Buchstabe, wenn er auf den Bildschirm "gemalt" wird, genau acht Grafikzeilen hoch ist: 25 * 8 = 200.

Demgegenüber ist die X-Auflösung wieder vom Grafikmodus abhängig: In Modus 0 sind es160 verschiedene Punkte, in Modus 1 320 und in Modus 2 640 Punkte. Daraus ergibt sich, daß in allen drei Modi die Buchstaben so auf den Bildschirm gezeichnet werden, daß sie genau acht Punkte breit sind:

80 * 8 = 640 bzw. 40 * 8 = 320 bzw. 20 * 8 = 160

Ein Buchstabe entspricht also immer einem Kästchen von 8 * 8 Punkten auf dem Bildschirm. Abhängig vom eingestellten Modus sind diese Kästchen nur unterschiedlich breit.

Die Farben

Der Schneider CPC bietet die Möglichkeit, den einzelnen Farbnummern erst während der Darstellung eine Farbe zuzuordnen. Deshalb ist für diese Farbnummern auch die Bezeichnung "Tinte" (Ink) üblich.

Auf dem Bildschirm wird mit unterschiedlichen Tinten gezeichnet. Unabhängig davon muß noch festgelegt werden, welche Farben die jeweiligen Tinten haben sollen. Ändert man eine Farbzuordnung, so werden alle Punkte, die mit der entsprechenden Tinte gezeichnet wurden, schlagartig mit der neuen Farbe dargestellt.

Diese Eigenschaft wird im CPC genutzt, um Tinten blinken zu lassen. Man ordnet einer Tinte nicht nur eine, sondern gleich zwei Farben zu. Eine spezielle Routine des Betriebssystems sorgt dafür, daß den Tinten im Rhythmus eines Taktgebers abwechselnd die eine und die andere Farbe zugeordnet wird: Alle Tinten, denen zwei verschiedene Farben zugeordnet wurden, blinken.

Insgesamt hat man die Wahl zwischen 27 verschiedenen Farben. Es können aber nicht alle Farben gleichzeitig dargestellt werden. Dabei liegt das Problem nicht so sehr bei den Farben, sondern bei den Tinten.

Wenn bisher auch immer gesagt wurde, daß beispielsweise im Modus 1 nur vier verschiedene Farben darstellbar sind, so ist das sprachlich etwas ungenau. Man ist nicht auf vier Farben beschränkt, sondern auf vier Tinten. Jeder Tinte kann jede der 27 verschiedenen Farben zugeordnet werden. Dadurch hat man auch im Modus 2 alle Farben zur Verfügung. Da man hier aber nur zwei Tinten benutzen kann, kann man auch immer nur zwei Farben gleichzeitig darstellen.

Die Tastatur

Die CPCs sind englische Produkte. Die Firma Schneider fungiert nur als Distributor für den deutschen Markt. Aus diesem Grund haben die CPCs auch eine englische Tastatur nach der ASCII-Norm: Die Tasten X und Y sind im Vergleich zur deutschen Belegung vertauscht, und die deutschen Spezialzeichen existieren nicht: §, ß und die Umlaute ä, ö und ü. Diese Zeichen haben nach der ASCII-Norm Codes, die den englischen Sonderzeichen [, \,], {, |, }, @ und ‘ entsprechen. Nur diese Zeichen sind zunächst über die Tastatur eingebbar.

Trotzdem kann man auch deutsche Zeichen ausdrucken: Auf der Tastatur kann man zur Not mit Abziehbildchen nachhelfen. Viel interessanter ist aber, daß auch auf dem Bildschirm deutsche Umlaute darstellbar sind. Beim Schneider CPC kann man ja das Aussehen jedes Zeichens verändern. Was hindert Sie also daran, aus einem [ein Ä zu machen?

Beläßt man es beim "Face Lifting", also dabei, das Aussehen eines [in ein Ä zu wandeln, kann man diese Zeichen sogar mit einem Drucker ausgeben: Der Drucker muß dazu nur auf den deutschen Zeichensatz umgestellt werden, was bei vielen Druckern sogar der Normalzustand ist, wenn diese verkauft werden.

Dann kann man natürlich die englischen Sonderzeichen nicht mehr so ohne weiteres ausdrucken. Statt der eckigen Klammer erhält man auf dem Papier immer ein Ä. Wie Sie aber vielleicht gemerkt haben, ist das Aussehen gar nicht so wichtig. Interessant ist der Code eines Zeichens: Bei der "eckigen Klammer auf" ist das 91. Ob das Zeichen mit dem Code 91 nun wie [oder wie Ä aussieht, kann man selbst festlegen. Beides zugleich ist aber nur schwer realisierbar.

Ein möglicher Ausweg wäre, auch die Belegung der Tasten zu verändern; also nicht das Aussehen eines Zeichens, sondern den Zeichencode, den die Taste erzeugt, wenn darauf gedrückt wird. Dazu sucht man sich aus der Palette der verspielten Grafikzeichen, die der Schneider CPC bereithält, einige aus, die man nicht benötigt, definiert deren Aussehen in Ä und Ü um und belegt eine geeignete Taste mit diesem Zeichen. Jetzt erhält man die englischen und die deutschen Sonderzeichen gleichzeitig. Die deutschen haben nur einen falschen Code.

Bei dieser Lösung wird der Text aber nicht richtig ausgedruckt. Die Zeichendarstellung des Druckers wurde damit ja nicht geändert. Dazu muß jedes Zeichen, das ausgedruckt werden soll, abgefangen und darauf kontrolliert werden, ob es sich um ein deutsches Sonderzeichen mit falschem Code handelt.

Immer wenn man ein Ä drucken will, muß der Drucker auf den deutschen Zeichensatz umgestellt und erst dann der Code für [oder Ä gesendet werden. Das Problem der Umlaute ist also lösbar.

Die Joysticks sind bei den CPC-Rechnern, rein physikalisch gesehen, an die Tastatur angeschlossen. Die Tasten der Joysticks, auch die "Richtungen", können wie Tasten der Tastatur behandelt werden. Der zweite Joystick ist dabei sogar völlig identisch mit sechs Tasten der Tastatur. Man kann also gar nicht unterscheiden, ob hier der Joystick betätigt oder eine Taste auf der Tastatur gedrückt wurde. Joystick Nummer eins hat dagegen eigene Tastenpositionen.

Alle Tasten können in BASIC mit INKEY direkt abgefragt werden. Komfortabler, zumindest bei der Texteingabe, ist INKEY$. Hierbei erhält man nicht die Information darüber, ob eine Taste gedrückt ist, sondern welche Zeichen in der Zwischenzeit durch Tastendrücke erzeugt wurden.

Die für die Tastatur zuständige Abteilung des Schneider-Betriebssystems verfügt sogar über eine Warteschlange für Tasteneingaben, so daß zeitweise schneller geschrieben werden kann, als das laufende Programm die Zeichen abarbeitet. Außerdem ist für alle Tasten einstellbar, ob diese ihren Anschlag automatisch wiederholen dürfen, wenn man sie längere Zeit gedrückt hält (Repeat-Funktion).

Weitere, veränderliche Tabellen enthalten die Tastenübersetzung: Hier ist gespeichert, welches Zeichen eine Taste erzeugen soll, wenn man sie drückt, welches Zeichen mit [SHIFT] und welches mit [CTRL] zusammen. Die Tastenbelegung ist also, wie bei den Umlauten bereits erwähnt, frei wählbar.

Die Tastatur-Software bietet sogar noch die Möglichkeit, speziellen Sonderzeichen, den Erweiterungszeichen, ganze Texte zuzuordnen. Oft benötigte Befehle können einem solchen Zeichen zugeordnet und dieses dann auf eine beliebige Taste gelegt werden. Immer, wenn man auf eine solche Taste drückt, erscheint automatisch ein ganzes Wort, wodurch man sich beim Programmieren viel Tipparbeit sparen kann.

Die Tasten des Zehnerblocks sind schon standardmäßig mit Erweiterungszeichen belegt. Allerdings sind diese, bis auf CTRL + ENTER, nur mit den entsprechenden Ziffern definiert, so daß man davon zunächst einmal nichts merkt.

Der Sound

Für die Tonerzeugung wird im Schneider CPC ein spezielles IC eingesetzt, ein digitaler, programmierbarer Sound-Generator.

Dieses IC verfügt über drei getrennte Kanäle. Für jeden Kanal ist die Lautstärke und Frequenz des Tonsignals getrennt einstellbar. Im Schneider CPC werden die Kanäle jedoch wieder vermischt: Alle drei Kanäle zusammengefaßt, werden dem eingebauten Lautsprecher zugeleitet.

Für den Stereo-Ausgang, mit dem man die Geräusche über die Stereo-Anlage abspielen kann, werden die drei Kanäle in zwei Gruppen zusammengefaßt: Kanal A und die Hälfte von Kanal B gehen zum linken Lautsprecher, Kanal C und die andere Hälfte von B zum rechten Lautsprecher. Der Anschluß an eine Stereo-Anlage ist übrigens sehr empfehlenswert, da der im Schneider CPC eingebaute Lautsprecher bei der Wiedergabe tiefer Töne völlig überfordert ist.

Weiterhin verfügt das IC über einen Rauschgenerator, der in seiner Grundfrequenz eingestellt werden kann. Dieses Rauschen kann dann einem oder mehreren Kanälen zugemischt werden. Der Rauschgenerator ist dabei nicht nur für statistische Zwecke implementiert worden, sondern sehr nützlich: Schußgeräusche, aber auch Beckenklänge einer Schlagzeugbegleitung lassen sich nur mit seiner Hilfe realisieren.

Mit einem Hüllkurvengenerator ist es darüber hinaus auch möglich, einem Kanal eine veränderbare Lautstärke zuzuordnen. Der Hüllkurvengenerator verfügt dazu über acht verschiedene Hüllkurvenformen, die auch noch mit unterschiedlicher Geschwindigkeit ablaufen können. Hiermit ist es dann möglich, das Ausklingen eines Glockenklangs oder ein Vibrato zu programmieren.

Obwohl der Sound-Generator an sich schon sehr leistungsfähig ist, sind seine Möglichkeiten durch eine entsprechend ausgeklügelte Treiber-Software noch einmal erheblich erweitert worden.

So ist es im Schneider CPC ohne weiteres möglich, bis zu 15 verschiedene eigene Hüllkurven für die Lautstärke zu programmieren. Dabei kann eine Hüllkurve bis zu fünf Abschnitte haben, in denen jeweils Länge, Steilheit und Raster eines Lautstärkeanstiegs oder -abfalls festgelegt werden. Dasselbe ist noch einmal für die Frequenz möglich.

Mit wirkungsvollen Befehlen kann man dann Töne zum Sound-Chip senden. Die Software nimmt dabei pro Kanal bis zu vier Töne in einer Warteschlange auf. Damit ist es ein Leichtes, kurze Zeiten, in denen sich das Hauptprogramm nicht um die Tonerzeugung kümmern kann, zu überbrücken.

Mit einem Interrupt-Mechanismus kann darüber hinaus die Tonerzeugung komplett vom Hauptprogramm abgekoppelt werden, auch in BASIC! Hierbei

wird immer dann ein Unterprogramm aufgerufen, wenn in der Warteschlange eines Kanals ein Platz leer geworden ist, weil ein Ton fertig abgespielt wurde.

Da speziell bei diesem "Bedienen bei Bedarf" die Tonerzeugung der einzelnen Kanäle mit der Zeit aus dem Tritt geraten kann, bietet das Betriebssystem auch noch einige Synchronisierungshilfen an. So kann man, quasi als Notbremse, alle Warteschlangen leeren und den nächsten Ton direkt anspielen. Dann kann man die Tonausgabe "einfrieren" und wieder "auftauen". Und man kann die einzelnen Kanäle mit einer Rendezvous-Technik synchronisieren. Kommt dann ein Ton um eine Hundertstelsekunde früher als sein Partner in einem anderen Kanal, so wartet er, bis auch dieser angespielt wird, ohne daß in seinem Kanal ein Ton erzeugt wird.

Die Massenspeicher

Beim Schneider CPC 464 ist ein Kassettenrecorder eingebaut. Für den Zweck, Programme zu speichern und wieder in den Computer zu laden, ist er durchaus ausreichend. Schwierig wird es erst, wenn man Ambitionen hat, ein schnelleres Aufzeichnungsverfahren zu benutzen. Besitzer eines CPC 664 oder 6128 haben da weniger Probleme: Sie können ihren HIFI-Rekorder anschließen.

Da die Speicherroutinen jedoch mit einer Motorsteuerung arbeiten, muß man unbedingt darauf achten, daß man einen Kassettenrecorder mit Remote-Eingang hat. Über diesen Eingang kann dann der Schneider CPC den Laufwerksmotor des Kassettenrecorders ein- und ausschalten. Andernfalls müßten Sie das von Hand erledigen, was eine sehr nervenaufreibende Tätigkeit sein kann.

Das liegt daran, daß der CPC seine Programme nicht in einem Stück, sondern scheibchenweise speichert. Dieses Verfahren hat den Vorteil, daß man eine Datei, die man speichern oder bearbeiten will, nicht komplett im Speicher halten muß. Man eröffnet eine Datei für die Ausgabe und schreibt dann nach und nach, so wie die Ergebnisse anfallen, die Daten in diese Datei.

Hinter dieser abstrakten Formulierung verbirgt sich, daß man die Zeichen, die man "in die Datei schreiben" will, an das Betriebssystem übergibt. Dieses überträgt das Zeichen erst einmal in einen Puffer, und wenn dieser voll ist, schreibt es den Puffer auf die Kassette. Kommen noch mehr Daten nach, ergeben sich eben entsprechend mehr Blöcke.

Die Speichergeschwindigkeit ist dabei in weiten Grenzen wählbar: Ca. 700 bis 2500 Baud verkraftet die Software; bei anderen Aufzeichnungsverfahren sind aber bis zu 7000 Baud möglich, wenn die Elektronik mitspielt. Von BASIC aus kann man aber nur zwischen zwei verschiedenen Geschwindigkeiten wählen: 1000 oder 2000 Baud. Dabei ist die Aufzeichnung mit 2000 Baud zwar doppelt so schnell, aber leider auch nicht mehr ganz so sicher.

Die Schneider CPC 664 und 6128 haben als Massenspeicher serienmäßig ein 3-Zoll-Laufwerk eingebaut. Auch der CPC 464 läßt sich damit nachrüsten. Pro Diskette können bis zu 360000 Bytes (Buchstaben) gespeichert werden. Auf jeder Diskettenseite sind das 180000 Bytes.

Pro Diskettenseite entspricht das etwa 60 eng beschriebenen Seiten Schreibmaschinenpapier. Auf die ganze Diskette gehen insgesamt 120 Seiten, aber gleichzeitig zugreifen kann man nur auf eine Diskettenseite! Die 3-Zoll-Laufwerke sind nämlich einseitig, auch wenn die Disketten beidseitig verwendbar sind.

Viele Programme, wie z.B. WordStar unter CP/M, können allerdings bei derartig wenig verfügbarem Speicherplatz nicht mit voller Leistung ablaufen. Zuerst einmal benötigt CP/M etwas Platz auf der Diskette, dann das Inhaltsverzeichnis, dann noch WordStar selbst mit seinen zwei Dialogdateien. Der verbliebene Rest steht der Textdatei zur Verfügung – leider auch nur zur Hälfte. Während man nämlich an einer Textdatei arbeitet, besteht immer noch eine "alte Version", des Textes als Sicherheit.

Bleibt als Ausweg nur der Erwerb eines zweiten Laufwerks. Dann stehen immerhin 360000 Bytes gleichzeitig zur Verfügung. Mehr Laufwerke können am AMSDOS-Controller nicht angeschlossen werden. In der Tat ist das ein wenig restriktiv: Nur zwei Laufwerke, diese nur einseitig mit 40 Spuren und dann auch noch das unübliche Format von 3 Zoll.

Wenn man kompatibel bleiben will, ist man auf AMSDOS mit mindestens einem 3-Zoll-Laufwerk angewiesen. Als Zweitlaufwerk kann man immerhin auch jedes andere Format nehmen. Das Laufwerk muß nur Shugart-kompatibel sein und ebenfalls einseitig mit 40 Spuren arbeiten. Ganz problemlos ist das zwar auch nicht. Man muß sich den Anschluß selbst basteln. Aber es werden auch einige 5.25-Zoll-Laufwerke angeboten, die direkt an den AMSTRAD- Controller passen.

Wem diese Restriktionen aber zu groß sind, der greift am besten gleich zu Controller und Laufwerken eines anderen Anbieters. VORTEX ist da wohl der bekannteste. Hier erhält man eine 5.25-Zoll-Doppelstation, mit der man

direkten Zugriff auf 1,4 Millionen Bytes Speicherplatz hat. Außerdem kann man auch hier noch eine 3-Zoll-Floppy anschließen, so daß man nachträglich umsteigen kann.

Der Speicher

Der im Schneider CPC verwendete Mikroprozessor, ein Z80, kann von Natur aus einen Gesamtspeicher von 2^{16} = 65536 Byte verwalten. Das ist für die heutigen Ansprüche etwas wenig. Und so ist der CPC 6128 bereits in seiner Grundausstattung mit dem doppelten an Schreib-Lesespeicher versehen.

Im Schneider CPC wird deshalb eine Technik angewandt, die mit "Bank Switching" bezeichnet wird. Dabei werden ganze Speicherbereiche zu Blöcken zusammengefaßt, die gemeinsam ein- oder ausgeblendet werden können. Dadurch kann man dann mehrere Speicherbereiche mit denselben Adressen ansprechen: Man muß nur dafür sorgen, daß in jedem Adreßbereich nur ein Block eingeblendet ist.

In den CPCs 464 und 664 ergibt sich folgendes Bild: Der Adreßbereich der CPU (des Mikroprozessors) ist logisch in vier Speicherviertel unterteilt. Die Blöcke, die ein- oder ausgeblendet werden, müssen immer ein ganzes Speicherviertel umfassen.

Der gesamte Adreßbereich der CPU ist zunächst einmal vollständig mit RAM (Arbeitsspeicher) belegt. Es gibt also vier RAM-Blöcke. Dabei enthält der obere RAM-Block normalerweise den Bildschirmspeicher.

Parallel zum untersten RAM-Block liegt ein ROM, also ein Speicher, dessen Inhalt fest programmiert ist und der nur gelesen werden kann. Hierin ist das Betriebssystem mit allen "elementaren" Funktionen des Computers enthalten.

Parallel zum obersten RAM-Block liegen alle weiteren ROMs. Im CPC 464 gibt es nur das ROM mit dem BASIC-Interpreter. Im CPC 664 und 6128 und im 464 mit Disketten-Controller liegt hier auch noch das ROM mit AMSDOS, dem Diskettenbetriebssystem.

Auch wenn man ROM-Module mit einer anderen Programmiersprache oder einem Spiel kauft, werden diese ROMs immer im obersten Adreßviertel eingeblendet.

Beim CPC 6128 sieht es im Prinzip genauso aus. Nur das zusätzliche RAM muß noch verteilt werden. Das wird bei normalen Anwendungen immer blockweise statt des zweiten normalen RAM-Blocks eingeblendet. Es kann aber auch komplett umgeschaltet werden, was aber meist Probleme verursacht, weil dann dem umschaltenden Programm "der Boden unter den Füßen weggezogen" wird.

Nach außen lassen sich die CPCs alle nur mit ROM-Modulen erweitern. Die Entwickler bei AMSTRAD haben nämlich den Kunstgriff begangen, alle Speicherschreibbefehle automatisch an das eingebaute RAM zu schicken. Versuchte man, externes RAM auch zu beschreiben, würden die Daten auch im interen RAM eingetragen. Da aber Speichererweiterungen normalerweise nur für das obere Speicherviertel vorgesehen sind, wo ja der Bildschirmspeicher liegt, würde man alles auf dem Bildschirm sehen, was man in externe RAMs schreibt.

BASIC

Das BASIC im Schneider CPC ist ein Interpreter wie die meisten anderen BASIC-Implementierungen auch. Ein Interpreter behandelt das BASIC-Programm wie einen Text: Er liest ihn und veranlaßt entsprechende Operationen. Dieses Verfahren ist vergleichsweise langsam, weil BASIC immer erst den Sinn einer Anweisung entschlüsseln muß, bevor es notwendige Operationen veranlassen kann.

Außerdem müssen die Referenzen während des Programmlaufs ausgewertet werden. Ein Sprung zu einer bestimmten Zeilennummer ist nur eine Referenz zu dieser Zeile. Genauso geht es auch mit den Variablen.

Bedeutend schneller sind da Compiler. Bei dieser "Technologie" wird der gesamte Programmtext nur einmal interpretiert und ein entsprechendes Maschinencode-Programm erzeugt. Das so gewonnene Programm kann dann von der CPU direkt ausgeführt werden. Der Nachteil dabei ist, daß man beim Programmlauf nicht mehr den Programmtext im Speicher hat. Die Fehlerbehandlung ist wesentlich erschwert. Solche Compilate sind meist auch viel länger als die Textdatei, aus der sie erzeugt wurden. Bei längeren Programmen kann da der Speicherplatz knapp werden.

Außerdem muß man nach der kleinsten Änderung im Programmtext das gesamte Programm neu compilieren und den so gewonnenen Maschinencode wieder neu einladen und starten.

Beim Locomotive BASIC, also dem "Dialekt", der im Schneider CPC eingebaut ist, werden die Vorzüge eines Interpreters mit denen eines Compilers verknüpft. Nach außen hin hat man zwar einen echten Interpreter. Für den Anwender unsichtbar compiliert er aber auch ein wenig: Die Adressen von Sprungzielen und Variablen werden bei jedem Durchlauf nämlich nur noch einmal bestimmt.

Wird eine Anweisung mehr als nur einmal durchlaufen, findet der Interpreter beim nächsten Mal bereits die richtige Adresse vor und muß nicht mehr lange suchen. Locomotive BASIC gehört deshalb zu dem schnellsten BASIC- Interpretern, die auf einem Homecomputer laufen.

Aber nicht nur die Geschwindigkeit des Locomotive BASIC ist beeindruckend, sondern auch die lange Liste an Befehlen, die es versteht. Der Schneider CPC hat ein sehr ausgefeiltes Betriebssystem, und Locomotive BASIC unterstützt fast alle Aspekte davon.

Die Eingaben von der Tastatur können in jeder nur denkbaren Form verarbeitet werden. Man kann testen, ob eine spezielle Taste gedrückt ist. Man kann aber auch Zeichen von der Tastatur "abholen", wobei man sich nicht mehr um die Tastenbelegung zu kümmern braucht. Außerdem kann man mit INPUT und LINE INPUT den Zeileneditor für einfache Eingaben benutzen. Die Tastenbelegung ist in jeder Beziehung frei definierbar; ebenso die Erweiterungszeichen, mit denen man auf Tastendruck ein ganzes Wort oder noch mehr erhält.

Auf dem Bildschirm kann man in jedem Modus arbeiten (im Gegensatz zu CP/M oder LOGO).

Man kann gleichzeitig in acht Textfenstern Text ausgeben. Dabei sind Textfenster Ausschnitte aus dem Bildschirm, die sich fast wieder wie kleine Bildschirme verhalten. Auch der Grafikausgabe ist ein Fenster zugeordnet.

Text kann in verschiedenen Farben ausgegeben werden, wobei sowohl die Vordergrund- als auch die Hintergrundfarbe wählbar ist. Zusätzlich gibt es den Transparentmodus, bei dem nur der Vordergrund, also die Punkte des Buchstabens selbst gezeichnet werden. Der Hintergrund bleibt dabei unverändert. Als Bonbon können Texte sogar an jeder beliebigen Grafikposition ausgegeben werden. Beim CPC 664 und 6128 ist es auch möglich, Buchstaben wieder vom Bildschirm zu lesen.

Zur Erstellung von Grafiken stehen Befehle zur Verfügung, die Punkte setzen oder ganze Linien zeichnen, in jeder nur möglichen Farbe und mit verschiedenen logischen Verknüpfungen mit der alten Farbe der übermalten Punkte.

Bei den CPC 664 und 6128 wurden die Grafikfähigkeiten sogar noch erheblich erweitert: Hier sind jetzt auch gestrichelte Linien möglich und eine schnelle Ausmal-Routine wurde implementiert.

Alle Funktionen der Tonausgabe werden von BASIC unterstützt: Drei Kanäle, Rauschen, je 15 Hüllkurven für Frequenz und Amplitude, Rendezvous- und Interrupt-Technik.

Gerade die Interrupt-Technik wurde auch für normale Unterbrechungen voll übernommen: Es gibt vier interne Uhren, die nach einer einstellbaren Zeit eine Unterbrechung des laufenden Programms auslösen und ein Unterprogramm ausführen. Diese Timer sind dabei mit AFTER für einen einmaligen Aufruf, mit EVERY aber auch für ständig wiederholte Unterbrechungen programmierbar.

Weitere Sonderfälle können zu Unterbrechungen führen: Fehler sind mit ON ERROR GOTO in eigenen Routinen abfangbar, außer den Diskettenfehlern beim CPC 464. Auch ein Druck auf die ESC-Taste (Break) kann zum Bearbeiten eines dafür vorgesehenen Unterprogramms führen. Nur beim 464 sind Breaks während INPUT oder LINE INPUT nicht abfangbar.

Zur Fehlersuche stehen die Befehle TRON und TROFF zur Verfügung. Nach TRON werden immer die Zeilennummern der gerade bearbeiteten Zeile im Fenster 0 ausgegeben. Leider ist diese Ausgabe nicht so ohne weiteres zu einem anderen Fenster oder gar dem Drucker umzuleiten. Dieselben Probleme hat man beim Ausgeben des Inhaltsverzeichnisses von Disketten oder Kassetten.

Auch größere Programmpakete können auf dem CPC entwickelt werden: Unterprogramme sind mit CHAIN oder CHAIN MERGE jederzeit nachladbar.

Bei den Variablen werden drei verschiedene Typen unterschieden: Integer (kleine ganze Zahlen), Real (große und Kommazahlen) und Strings (Zeichenketten). Für die Variablennamen sind dabei allgemeine Einstellungen möglich, um bestimmten Variablennamen aufgrund ihres ersten Buchstabens einem Typ zuzuordnen.

Der Typ läßt sich aber auch noch explizit durch einen Postfix, also einen Anhang am Variablennamen bestimmen: ! für Real, % für Integer und $ für Strings.

Diese Regelungen gelten übrigens auch für die selbstdefinierbaren Funktionen.

Zur Erstellung eines Programms steht eine Mischung aus Zeilen- und Bildschirmeditor zur Verfügung. Ein Text (im Normalfall eine Programmzeile) kann bis zu 255 Zeichen lang sein. Mit der COPY-Taste kann man Buchstaben von jeder Stelle des aktiven Textfensters in die Programmzeile übernehmen. Das ist sehr praktisch, wenn man größere Umstellungen innerhalb des Programms vornehmen will.

Zur Textaufbereitung stehen außerdem Befehle wie EDIT, AUTO, LIST und RENUM zur Verfügung. Mit EDIT übernimmt man eine existierende Programmzeile in den Zeileneditor, kann sie dort bearbeiten und wieder abspeichern. AUTO erspart das Eintippen der Zeilennummern, die nun automatisch erzeugt werden. Mit LIST kann man sich den aktuellen Inhalt der Programmdatei anzeigen lassen: auf dem Bildschirm, Drucker oder (manchmal sehr nützlich) auch in eine Datei auf Kassette oder Diskette hinein. Mit RENUM schließlich kann man die Zeilennummern wieder in ein gleichmäßiges Raster bringen. Das ist zum Beispiel sinnvoll, wenn durch ständiges Einfügen an einer Stelle keine Zeilennummern mehr verfügbar sind.

Auch eine komfortable Schnittstelle zu Maschinencode-Programmen wurde ins BASIC integriert: Mit CALL können Maschinencode-Routinen mit einer bekannten Anfangsadresse aufgerufen werden. Dabei können bis zu 32 Zahlenwerte als Argumente an den Maschinencode übergeben werden.

Eine spezielle Form der Maschinenprogramme sind die RSX-Befehlserweiterungen. Diese Programme sind meist zu mehreren in Programmpaketen zusammengefaßt und müssen initialisiert werden. Dann stehen sie dem Anwender aber mit einem Namen zur Verfügung. In BASIC werden solche RSX-Namen mit einem vorangestellten senkrechten Strich (|) gekennzeichnet. Auch hier können wieder bis zu 32 Zahlen an das Programm übergeben werden. In der Praxis wird man auf solche RSX-Erweiterungen zuerst beim Diskettenbetriebssystem AMSDOS stoßen. Das stellt seine zusätzlichen Funktionen nämlich alle über RSX-Erweiterungen zur Verfügung.

LOGO

Mit dem Kauf des ersten 3-Zoll-Diskettenlaufwerkes, bei den CPC 664 und 6128 also mit dem Kauf des Computers selbst, erhält man außer CP/M auch noch eine weitere Programmiersprache: LOGO, in einer Version von Digital Research.

LOGO ist eine strukturierte Programmiersprache. In LOGO gibt es kein Programm in dem Sinn wie in BASIC. Zunächst werden sogenannte *Prozeduren*

für die elementarsten Probleme definiert. Solche Prozeduren können sowohl Eingabe- als auch Ausgabeparameter haben, ähnlich den Funktionen in BASIC. Aufbauend auf den so definierten Prozeduren definiert man dann die "zweite Generation": Prozeduren die bereits kompliziertere Aufgaben übernehmen können. Das geht dann so lange weiter, bis man zum Schluß nur noch eine Prozedur definieren muß, die das gesamte Programm "beinhaltet", z.B.:

```
TO SPIEL
   TITELBILD
   SCHWIERIGKEITSSTUFE
   SPIELTEIL
   HISCORELISTE
   END
```

An diesem Beispiel sieht man auch schon, wie eine LOGO-Prozedur in etwa aufgebaut sein muß: Am TO erkennt man, daß die Definition einer Prozedur folgt. SPIEL ist der Name der Prozedur. Das Ende der Definition wird durch END angezeigt. Dazwischen stehen die Aufrufe von anderen bereits definierten Prozeduren.

Dabei sind natürlich auch Schleifen und Variablen möglich. Insbesondere ist es möglich, Zahlen, Variablen oder ähnliches als Parameter einer aufgerufenen Prozedur zu übergeben. Diese Prozedur kann nun ihre eigenen Variablen definieren, die denselben Namen haben können wie Variablen im Hauptprogramm. Dadurch fällt das leidige Problem weg, daß Variablen durch ein Unterprogramm unbeabsichtigt verändert werden. Zum Schluß kann eine Prozedur auch wieder Funktionswerte, also Ergebnisse an die rufende Prozedur zurückgeben. Die Prozedur SIN übernimmt beispielsweise als Eingabe einen Winkel und liefert als Ausgabe den Sinuswert.

Dadurch, daß eine Prozedur ihre eigenen (lokalen) Variablen definieren und sich sogar selbst aufrufen kann, sind rekursive Problemlösungen möglich.

Eine weitere Besonderheit von LOGO sind die sogenannten Listen. Mehrere Zahlen oder Zeichenketten können in einer Liste zusammengefaßt werden. Diese Liste wird dann einer Variablen zugeordnet. Listen ersetzen in LOGO teilweise die dimensionierten Felder, die LOGO nicht kennt.

Wer jedoch LOGO hört, denkt direkt auch an "TURTLE GRAPHIC". Diese "Schildkröten-Grafik" ist ein Zugeständnis der LOGO-Entwickler an die kindlichen Benutzer. LOGO kennt keinen koordinaten-, sondern einen vektororientierten Bildschirm.

Wer in BASIC gewöhnt ist, Punkte auf bestimmte (X,Y)-Koordinaten zu setzen und Linien zu solchen Punkten zu ziehen, muß in LOGO umdenken.

Gezeichnet wird ausschließlich von einem Symbol, das "Turtle", also Schildkröte, getauft wurde. Diese sitzt an einer bestimmten Stelle auf dem Bildschirm und schaut in eine bestimmte Richtung. Man kann man sich entscheiden, ob sie den Stift aufsetzen soll und welche Farbe der Stift haben soll. Da CPC-LOGO in Modus 1 arbeitet, können bis zu vier Farben gleichzeitig dargestellt werden.

Dann kann man die Schildkröte drehen, mit RT für rechts herum und LT für links herum. Und man kann die Schildkröte bewegen: z.B. mit FD für vorwärts oder BK für rückwärts. Ist der Zeichenstift aufgesetzt, zeichnet die Schildkröte eine Linie.

Ein gleichseitiges Dreieck erhält man so in LOGO ganz einfach mit folgender Befehlsfolge:

```
FD 100      100 Schritte vor
RT 120      um 120 Grad nach rechts drehen
FD 100
RT 120
FD 100
```

Da der Bildschirm jedoch (physikalisch gesehen) koordinatenorientiert ist, muß LOGO intern ständig mit Sinus und Cosinus rechnen, um dem Anwender so einfache Befehle zur Verfügung zu stellen. Das führt dann dazu, daß LOGO beim Zeichnen noch langsamer als sonst wird.

Da dem Anwender außerdem nur etwa 8400 Bytes für Prozeduren und Variablen zur Verfügung stehen, ergeben sich zusammen mit der langsamen Programmbearbeitung zwei schwerwiegende Hindernisse für eine ernsthafte Anwendung.

CP/M

Ebenfalls mit dem ersten Diskettenlaufwerk erwirbt man CP/M. Hierbei handelt es sich um kein Programm im eigentlichen Sinne. CP/M stammt von der Firma Digital Research und ist eine Abkürzung für: Control Program for Microprocessors.

CP/M schafft für alle Programme eine genau definierte "Umgebung". CP/M ist die Schnittstelle zwischen einem idealisierten, fiktiven Computer und dem Gerät, das tatsächlich bei ihnen auf dem Schreibtisch steht. Alle Programme, die unter CP/M laufen, wissen nicht, was für ein Computer das eigentlich ist.

Sie halten sich an die Ein- und Ausgabeschnittstellen, die CP/M bereithält. CP/M behandelt die Daten weiter, gibt sie zum Beispiel auf dem Bildschirm aus oder schreibt sie auf Diskette.

Dadurch, daß CP/M quasi einen genormten Computer simuliert, stehen jedem Anwender eine Vielzahl von Programmen zur Verfügung, die alle auf diesem genormten Computer, also unter CP/M, lauffähig sind.

Welche Anforderungen stellt CP/M an den Computer, was muß dieser mindestens können?

Am wichtigsten ist wohl der verwendete Mikroprozessor. Es kann ein 8080 oder auch ein Z80 sein, wie er im Schneider CPC benutzt wird. Dieser ist eine Weiterentwicklung des 8080 und verhält sich fast genauso wie dieser. Der Z80 kann nur noch etwas mehr: Er ist zum 8080 aufwärtskompatibel.

Für CP/M muß mindestens ein Diskettenlaufwerk vorhanden sein, denn CP/M ist ein Disketten-orientiertes Betriebssystem. Das geht sogar so weit, daß man CP/M immer erst von einer Diskette laden muß.

Leider hat CP/M auf dem Schneider CPC einige Probleme: Einmal die unüblichen 3-Zoll-Disketten, dann der relativ kleine, für CP/M-Programme frei verfügbare Speicherbereich (TPA) und die langsame Textausgabe auf dem Bildschirm. Textverarbeitungsprogramme sind oft endlos lange mit dem Bildschirmaufbau beschäftigt.

Das sind zwar alles keine Hinderungsgründe, schränken die Anwendungen jedoch oft erheblich ein. Glücklicherweise ist beim Schneider CPC wenigstens der Diskettenzugriff ziemlich flott.

Hat man CP/M mit |CPM gebootet, also von der Diskette geladen, so kann man ihm Befehle erteilen. Einige wenige kann CP/M selbst ausführen. In den meisten Fällen wird dadurch aber erst das entsprechende Programm von der Diskette in den Arbeitsspeicher geladen und aufgerufen.

Direkt verfügbare (residente) Befehle sind:

DIR	Gibt ein Inhaltsverzeichnis der Diskette aus
REN	Umbenennen einer Datei
ERA	Löschen einer Datei
TYPE	Gibt den Inhalt einer Datei auf dem Bildschirm aus
USER	Wählt eine andere, logische Benutzernummer aus
SAVE	Schreibt den angegebenen Teil des Programmspeichers (TPA) als lauffähiges CP/M-Programm auf die Diskette.

Alle anderen (transienten) Befehle müssen erst von der Diskette geladen werden und dort natürlich auch vorhanden sein. Diese Befehle erkennt man im Inhaltsverzeichnis an der Extension .COM.

Die wichtigsten transienten Befehle sind:

FORMAT	um Formatieren von Disketten
PIP	um Kopieren, Ein- und Ausgabe von Dateien
FILECOPY	um Kopieren einzelner Files (auch auf Disketten ohne CP/M)
DISCCOPY	um Kopieren ganzer Disketten (oder COPYDISC bei zwei Laufwerken)
ED	Editor für Textdateien

Kapitel 1

Grundlagen

Datenspeicherung und Datenstrukturen

Bei jedem Programm kann man drei Ebenen im Umgang mit den Daten ausmachen: die Datenverarbeitung, bei der der Aufbau der einzelnen Daten bekannt sein muß, der Datenzugriff und die Datenspeicherung. Vor allem die letzten beiden Gruppen werden in der Informatik streng unterschieden.

"Datenstrukturen" beschreiben den Zugriff auf einen Datenspeicher, also auf welche Weise Daten hinein- und wieder herauskommen dürfen. Beispiele sind:

- Stacks (Datenstapel, LIFO)
- Queues (Warteschlangen, FIFO)
- Fields (Felder, wahlfreier Zugriff)
- Trees (Bäume, verästelter, hierarchischer Zugriff)

Mit "Datenspeicherung" bezeichnet man die physikalische Darstellung der Datenelemente, vor allem, wie im Speicher des Computers der Zusammenhalt zwischen den einzelnen Speicherelementen realisiert ist. Mögliche Formen sind:

- Arrays (gepackte Daten) und
- Chains (verkettete Daten)

Die einzelnen Datenelemente eines Arrays folgen im Speicher dicht an dicht. Der Zusammenhalt und der gezielte Zugriff auf einzelne Elemente wird durch ihre Lage innerhalb des vom Array beanspruchten Speicherbereichs bestimmt.

Bei verketteten Listen (Chains) können die einzelnen Datenelemente vollkommen willkürlich im Speicher verteilt sein. Der Zusammenhalt wird über Zeiger realisiert. Jedes Datenelement hat mindestens einen Zeiger, der auf das nächste Datum zeigt. Dadurch kann man sich von einem festen Anfang bis zum Ende durch die gesamte Datei durcharbeiten.

RECORDS

Die einzelnen Datenelemente werden "Records" genannt. Sie stellen innerhalb der Datei die kleinste Einheit dar. Ob eine Datei aus Fließkommazahlen besteht, Bits oder ganzen Datensätzen mit Name, Straße und Wohnort: Die (physikalisch meist noch weiter teilbaren) Zahlen oder Datensätze sind die Records.

Records können sich in ihrem "Typ" unterscheiden:

- Flags (Bits)
- Characters (Bytes)
- Integer-Zahlen (Words)
- Real-Zahlen (5 Bytes)
- Strings (Zeichenketten beliebiger Länge)
- Listen (Zusammenfassungen mehrerer Elemente auch unterschiedlichen Typs)
- Datensätze

Viel wichtiger ist aber die Unterscheidung zweier grundverschiedener Kategorien:

- FLR Fixed Length Records (Records mit konstanter Länge)
- VLR Variable Length Records (Records mit veränderlicher Länge)

Die einzelnen Record-Typen lassen sich dabei meist recht eindeutig der einen oder anderen Kategorie zuordnen: Bits und Integer-Zahlen sind beispielsweise FLR, Strings und Listen sind VLR. Dabei ist diese Zuordnung jedoch nicht eindeutig und kann von Fall zu Fall unterschiedlich sein. So ist es beispielsweise denkbar, für Strings eine feste Record-Länge vorzugeben, die nur eben nicht immer vollständig ausgenutzt wird. Andererseits können aber auch Zahlen verschieden lang sein, wenn man sie beispielsweise als ASCII-Sequenz in eine Disketten- oder Kassettendatei schreibt.

Auf jeden Fall beeinflußt die Kategorie der Records, ob FLR oder VLR, in aller Regel ganz entscheidend die Datenspeicherung.

So sind Fixed Lenght Records geradezu prädestiniert für Arrays und VLRs für verkettete Listen. Dies ist natürlich auch wieder nicht als 100%ige Trennung anzusehen. Arrays und verkettete Listen haben unterschiedliche Vorteile, die man durchaus auch für die jeweils andere Kategorie benötigen kann.

ARRAYS

Felder als FLR-Array

Bei Arrays aus Fixed Length Records (FLR) ist beispielsweise der Zugriff auf die einzelnen Records sehr schnell. Felder mit wahlfreiem Zugriff werden fast ausschließlich als FLR-Array realisiert! Die exakte Lage eines Records läßt sich nämlich aus seinen Indizes, der Länge der einzelnen Records und der Startadresse (Basis) des Arrays berechnen:

Rec.Adr. = Arraybasis + Rec.Len. * (i0 + im0*i1 + im0*im1*i2 + ...)

i0, i1 usw. stellen hierbei die Indizes der verschiedenen Array-Dimensionen dar und im0, im1 usw. die Größe der einzelnen Dimensionen. i0 bzw. im0 entsprechen dabei dem innersten Index.

Beispiel:

Die Zahlenfelder, die von BASIC angelegt werden, sind FLR-Arrays. Deshalb kann man ja auch auf jede beliebige Zahl innerhalb des Feldes sofort zugreifen. Mit dem folgenden Beispiel wird ein Integer-Feld dimensioniert:

```
DIM a% (3, 5, 2)
       ↑  ↑  ↑
      im0 |  im2
          im3
```

Das Feld hat drei verschiedene Dimensionen, erkennbar an den drei Maximal-Indizes. Die Größe der ersten Dimension ist 4 (und nicht 3), weil insgesamt vier Datenplätze in dieser Richtung ansprechbar sind: 0, 1, 2 und 3. Die Größe der zweiten Dimension ist 6, die der dritten ist 3.

Die Record-Länge in einem Integer-Array beträgt zwei Bytes. Die Array-Basis ist die Adresse des Records a%(0,0,0), und der innerste Index ist im Schneider-BASIC der erste Index innerhalb der Klammer.

Die Adresse des Records a%(3,4,1) läßt sich dann so berechnen:

```
@a% (3, 4, 1) = @a% (0, 0, 0,) + 2 *(3 + 4 *4 + 4*6*1)
      ↑  ↑  ↑                      ↑  ↑  ↑  ↑  ↑  ↑
      i0 i1 i2                     i0 im0 i1 im0 | i2
                                                im1
```

Dieses BASIC-Programm zeigt das geordnete Aufeinanderfolgen der einzelnen Feldelemente:

```
10 DIM a%(3,5,2)
20 FOR k=0 TO 2
30 FOR j=0 TO 5
40 FOR i=0 TO 3
50 PRINT @a%(i,j,k)
60 NEXT i,j,k
```

Die ausgedruckten Adressen folgen genau im Zweierabstand aufeinander. Der innerste Laufindex i entspricht der innersten Dimension von a%(). Ist die innerste Dimension einmal komplett durchlaufen, so wird der Index der mittleren Dimension geändert, und es folgt wieder eine Sequenz, bei der sich nur der innerste Index ändert.

ARRAYS AUS VL-RECORDS

Sollen Variable Length Records in einem Array gespeichert werden, so kann natürlich nicht mehr von einer einheitlichen Record-Länge ausgegangen werden. Mithin ist diese Formel sinnlos. Um hier ein spezielles Array-Element zu finden, muß man sich vom ersten Record an durcharbeiten. Dazu müssen die VL-Records mit einer geeigneten Kennung voneinander getrennt werden. Der Zugriff auf einen speziellen Record wird hier ziemlich zeitintensiv.

Jeder BASIC-Interpreter speichert das Programm aber als einen Array von VL-Records in seinem Speicher ab! Hier gibt es auch kaum eine andere Wahlmöglichkeit. Die einzelnen Programmzeilen können nun einmal unterschiedlich lang sein und werden im Speicher direkt aufeinanderfolgend abgespeichert. Bei jedem Sprung oder Unterprogramm-Aufruf muß der Interpreter dann den gewünschten Record (hier eine Programmzeile) erst suchen.

Dazu kommt noch, daß der Variablenbereich ebenso aufgebaut ist. Ein normaler BASIC-Interpreter verbringt also viel Zeit mit der Suche nach VLRs in diesen beiden Arrays. Das Locomotive-BASIC im Schneider CPC wurde hier glücklicherweise erheblich beschleunigt.

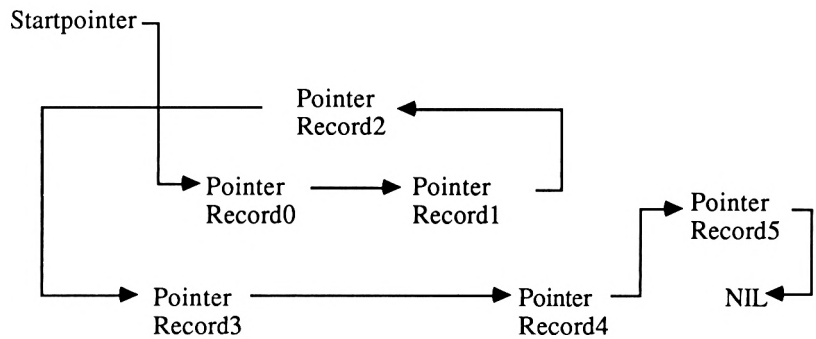
Ein grundsätzlicher Nachteil von Arrays wird bei der Betrachtung dieser BASIC-Arrays deutlich: Wird eine BASIC-Zeile gelöscht oder eingefügt, so müssen alle nachfolgenden Zeilen verschoben werden. Bei den BASIC-Zeilen ist das nicht weiter schlimm. Die Zehntelsekunde, die der Interpreter dafür benötigt, bemerkt man kaum.

Bei VL-Records kommt noch hinzu, daß dieses Verschieben nicht nur stattfindet, wenn ein Record eingefügt oder entfernt wird. Auch wenn ein Record (z.B. eine BASIC-Zeile) seine Länge ändert, müssen alle nachfolgenden Records (Zeilen) entsprechend verschoben werden. Würden auch die Strings im Variablenbereich so behandelt, könnte von Verarbeitungsgeschwindigkeit in BASIC wohl kaum mehr die Rede sein.

CHAINS

Bei verketteten Listen (Chains) werden die einzelnen Records im Speicher nicht aufeinanderfolgend abgelegt. Ihre Lage ist überhaupt völlig beliebig! Um den Zusammenhalt einer Datei zu wahren, wird jeder Record gemeinsam mit einem oder mehreren Zeigern abgespeichert, die auf das folgende, vorhergehende oder ein sonstwie benachbartes Listenelement zeigen.

Einfache verkettete Liste:



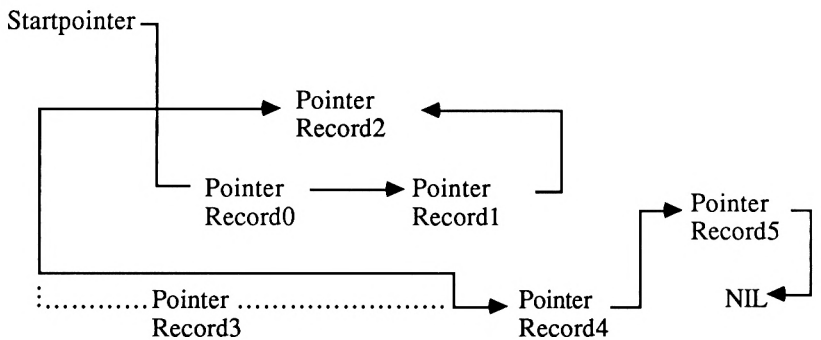
Diese Grafik zeigt eine einfache verkettete Liste mit sechs Elementen. Wie man sieht, können die einzelnen Elemente überall im Speicher liegen und werden nur über die Pointer miteinander verbunden. Die zeigen in diesem Beispiel immer nur auf das nächstfolgende Element. Zwei weitere wichtige Eigenschaften einer solchen Verkettung sind erkennbar: Zum einen muß der Startpunkt, also die Adresse des ersten Kettenelements, bekannt sein und zum anderen muß das Kettenende durch einen speziellen Pointer gekennzeichnet werden. Bei Z80-Systemen verwendet man dazu sehr oft einfach die Adresse &0000. In diesem Bereich liegen ja die Restarts, und es ist normalerweise unmöglich, hier Daten abzuspeichern. Die Bezeichnung "NIL" für diesen Endpointer steht dabei für "Not In List".

Um also ein spezielles Datenelement aufzufinden, muß man auch hier, egal ob man FLRs oder VLRs verkettet hat, beim ersten Element anfangen und sich mittels der Kettungspointer von einem zum anderen Record durchharbeiten, bis man das gewünschte Datenelement erreicht hat. Verkettete Listen werden deshalb fast nie eingesetzt, wenn ein wahlfreier Zugriff auf die einzelnen Elemente eines Feldes gewünscht ist.

Verkettete Listen haben, bis auf den erhöhten Speicherplatzverbrauch für die Kettungspointer, ansonsten aber fast nur noch Vorteile: Die Anzahl der Records in einer Datei kann beliebig schwanken. In einer Dateiverwaltung ist folgender Fall denkbar: Eine Datei wird eingerichtet und umfaßt noch kein einziges Element. Es wird praktisch nur ein Startpointer eingerichtet, der gleichzeitig Endpointer ist. Er zeigt auf NIL. Eine solche Liste verbraucht noch keinen Speicherplatz!

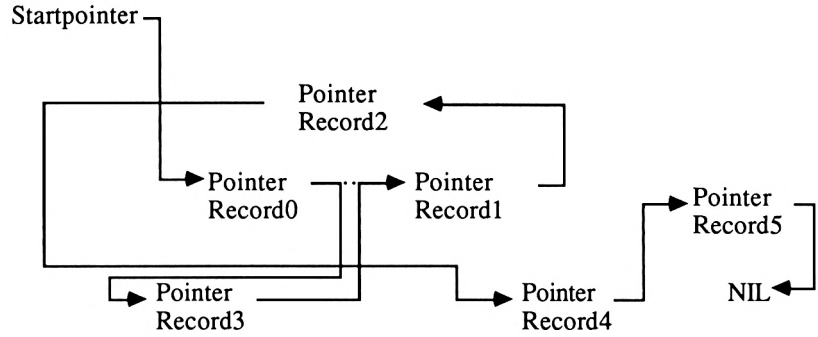
Erst wenn nach und nach Datensätze (Records) eingegeben werden, beanspruchen diese auch Platz. Das Einfügen und Wieder-Aushängen von einzelnen Datensätzen ist äußerst komfortabel: Man braucht nicht alle nachfolgenden Records um einen Platz zu verschieben, sondern verändert nur einen Pointer.

Aushängen eines Records aus einer Chain:



Record3 aus dem vorherigen Bild wurde einfach dadurch ausgehängt, daß der Pointer in Record2 verstellt wurde. Dieser zeigt jetzt auf Record4, wodurch Record3 nicht mehr erreichbar ist. Der Pointer von Record3 zeigt möglicherweise auch weiterhin auf Record4, was aber ohne Bedeutung für diese Kette ist.

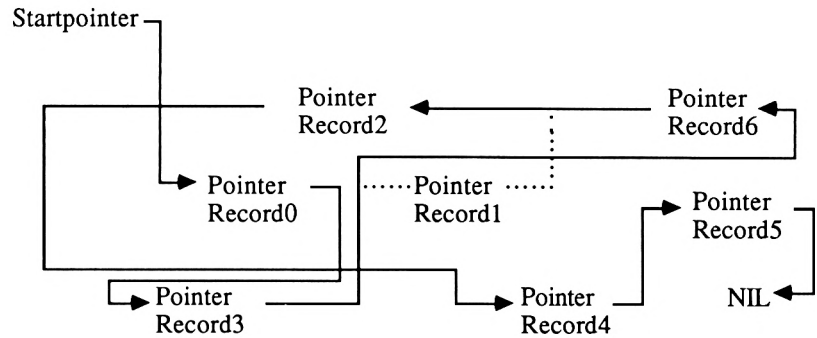
Einhängen eines Records in eine Chain:



In diesem Beispiel wurde der ehemalige Record3 zwischen Record0 und Record1 eingehängt. Dazu mußte einfach der Pointer, der von 0 nach 1 zeigte, "aufgetrennt" werden. Pointer0 zeigt nun auf Pointer3 und dieser wiederum auf Pointer1.

Beim Einfügen oder Aushängen verändern sich natürlich die Platznummern aller nachfolgenden Records. Das kann aber vermieden werden, indem ein neuer Record eben nicht nur eingehängt, sondern ein Listenelement, dessen Platz er einnehmen soll, aus der Kette entfernt wird:

Ersetzen eines Records in einer Chain:



In diesem Beispiel wurde Record1 ausgehängt und ersatzweise Record6 eingefügt. Record1 wurde also durch Record6 ersetzt. Dadurch rücken die nachfolgenden Elemente keinen Platz vor oder zurück.

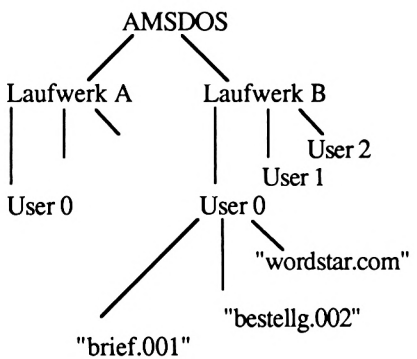
TREES

"Bäume" stellen eine Datenstruktur dar. Hier ist der Zugriff auf einzelne Daten meist hierarchisch gegliedert. Man fängt bei der "Wurzel" (root) an und kann sich dann über mehrere Stufen zum gewünschten Datenelement hinbewegen. Auf jeder Stufe "verästelt" sich der Baum. Man hat dann mehrere Möglichkeiten, sich für den Pfad zu entscheiden, der zum gewünschten Datenelement führt.

Sub-Directories

Eine sehr sinnvolle Anwendung für solche Hierarchien sind Sub-Directories, die man bei manchen Disketten-Betriebssystemen auf einer Diskette anlegen kann. Beim ATARI ST werden sie beispielsweise "Ordner" genannt; dahinter verbirgt sich aber genau dieses Konzept. Bei den geringen Speicherkapazitäten der 3-Zoll-Floppies sind derartige Spielereien noch nicht unbedingt nötig, immerhin kann man seine Diskette aber auch hier in mehrere USER-Bereiche einteilen.

Sinnvoll sind solche Unter-Inhaltsverzeichnisse vor allem dann, wenn ein Speicher mit mehreren Megabytes zur Verfügung steht, also eine Festplatte etwa. Der Zugriff auf eine bestimmte Datei unter AMSDOS könnte immerhin auch schon baumförmig dargestellt werden:



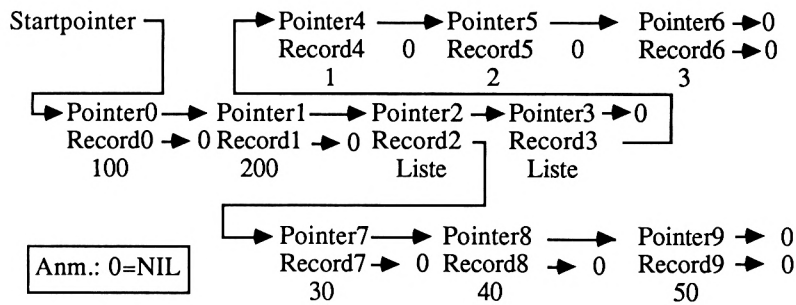
Um den Brief zu laden, muß man AMSDOS bewegen, auf Laufwerk B im User-Bereich 0 nachzuschauen und dort die Datei "brief.001" zu laden. AMSDOS ist die Wurzel, und der Baum wächst typischerweise nach unten.

Listen

Auch die Listen von LOGO lassen sich leicht mit einer baumartig verzweigten Kette realisieren. Jedem Datenelement muß man dabei (mindestens) zwei Pointer zuordnen: einen, mit dem man ein Element innerhalb einer übergeordneten Liste einreiht und einen, der eventuell auf eine eigene Liste zeigt, die durch dieses Element "benannt" ist:

Liste in LOGO: [100 200 [30 40 50] [1 2 3]]

Diese Liste enthält zwei Unterlisten: [30 40 50] und [1 2 3]. Das folgende Beispiel zeigt, wie man diese verschachtelten Listen im Speicher darstellen kann:



Es gibt aber noch viele andere mögliche Formen für verkettete Listen. Man kann auch Ringspeicher realisieren, indem man in den letzten Pointer nicht NIL, sondern wieder einen Zeiger auf das erste Element der Liste einträgt. Man kann aber auch die Records in einer verketteten Liste über je einen Vor- und Rückwärtszeiger verknüpfen, um sich innerhalb der Liste in beiden Richtungen bewegen zu können. Nicht nur baumartige Strukturen sondern auch mehrdimensionale Felder sind realisierbar. Unter praktischen Gesichtspunkten sind die aber fast ausschließlich den Arrays vorbehalten.

Verkettete Listen im CPC

Das Betriebssystem des Schneider CPC verwendet verkettete Listen hauptsächlich im Kernel: Die Interrupt-verursachenden Listen sind allesamt Chains:

- Ticker Chain
- Fast Ticker Chain
- Frame Flyback Chain

Auch die Listen, in die die gekickten Ereignisse bis zu ihrer Bearbeitung eingereiht werden, sind Chains:

- asynchronous pending queue
- synchronous pending queue

Bei der letzteren kommt als eine Variante noch hinzu, daß hier die einzelnen Einträge eine Priorität haben und vom Kernel entsprechend eingereiht werden.

Außerdem sind auch noch die Tabellen für die externen Kommandos (RSX) über verkettete Listen verknüpft, wofür man bei jedem Aufruf von KL LOG EXT vier Bytes des freien Speichers zur Verfügung stellen muß.

STACKS: LAST IN – FIRST OUT

Die erste Datenstruktur, mit der man zu tun hat, sind wohl die Felder mit wahlfreiem Zugriff, wie sie BASIC bereithält. Als nächstes kommt in aller Regel der Stack.

Stacks sind *Datenstapel*, und diese Bezeichnung deutet an, wie sie funktionieren: Auf den wie auch immer realisierten Stapel kann man Daten ablegen, um sie zu einem späteren Zeitpunkt wieder herunterzunehmen. Dabei ist der Zugriff immer auf das oberste Element des Stapels beschränkt.

Das Datenelement, das man zuletzt auf dem Stapel abgelegt hat, muß als erstes wieder heruntergenommen werden. Daher rührt auch die andere Bezeichnung, die für Stacks gebräuchlich ist:

LIFO-Speicher : Last In – First Out

Ganz besonders die verschiedenen Unterprogrammtechniken profitieren davon. So können in Stacks die Rücksprungadressen für Unterprogramm-Aufrufe gespeichert, Argumente an Funktionen (also: Unterprogramme) übergeben und von diesen wieder übernommen und lokale Variablenbereiche realisiert werden.

Ein Stack als FLR-Array

Jede CPU verfügt über einen solchen Stapel. Beim Z80 können darauf normalerweise nur 2-Byte-Worte (Words) abgelegt werden. Alle im Stapel enthaltenen Elemente liegen direkt hintereinander. Es ist also ein Array aus Fixed

Length Records. Nur die Lage der Stapelspitze ist dem Prozessor bekannt. Der Stapelboden (Base), also die Speicheradresse des letzten Stapelelements, wäre zwar auch ganz interessant, ist aber nicht so wichtig. Es ist dem Prozessor also jederzeit möglich, mehr Worte vom Stack zu lesen, als er vorher drauf abgelegt hat. Kommt so etwas tatsächlich einmal vor, kann man ziemlich sicher sein, daß im Programm ein Fehler steckt.

Das laufende Programm muß sich den verfügbaren Speicherbereich aufteilen und dabei dem Stack einen Speicherbereich reservieren, der für den maximalen Bedarf, der irgendwo im Programm entstehen kann, ausreicht. Denn dem Prozessor ist nicht nur unbekannt, wo der Stapel anfängt, er weiß auch nicht, wie hoch er höchstens werden darf.

Das einzige, was der Prozessor kann, ist Daten drauflegen und wieder runterholen. Die Stelle, an der er das tut, die Stapelspitze, wird durch ein speziell dafür vorgesehenes Doppelregister, den "Stackpointer" (SP) adressiert.

Der Stapel des Z80 hat dabei die Eigenschaft, nach unten zu wachsen. Jedesmal, wenn ein neues Wort auf dem Stapel abgelegt wird, wird der Stackpointer um zwei Adressen erniedrigt. Nimmt man umgekehrt ein Datenelement wieder weg, wird der Stackpointer um zwei Adressen heraufgesetzt. Der Stackpointer zeigt dabei immer auf das unterste Byte (LSB – Least Significant Byte) des letzten, im Stapel enthaltenen Wortes.

Die wichtigsten Operationen, die den Stapel wachsen lassen, sind:

1. Unterprogramm-Aufrufe (CALL) und
2. Register-Speicherbefehle (PUSH).

Umgekehrt nehmen die folgenden Befehle wieder Werte vom Stapel herunter:

1. Unterprogramm-Rücksprung (RET)
2. Register-Ladebefehle (POP)

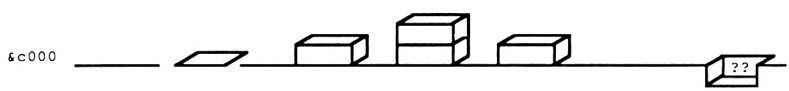
Die genaue Funktionsweise des Stapels bei der Unterprogramm-Behandlung wird erst bei den Programmstrukturen behandelt.

Mit den Befehlen PUSH und POP können Registerinhalte kurzzeitig gerettet (PUSH) und wieder restauriert werden (POP), beispielsweise vor und nach einem Unterprogramm-Aufruf, der wichtige Register verändern könnte.

Das folgende Beispiel zeigt die Errichtung des Maschinenstapels und einige Aktionen darauf. Zuerst als Assembler-Programm und dann in den Grafiken die zugehörigen Darstellungen des Stapels selbst:

```
10   LD    SP,#C000   ; Stapelspitze neu setzen. Annahme:
                      ; Der Stapel soll jetzt neu
                      ; und leer sein. Stapelboden ist also
                     ; &C000.
20   LD    HL,#1234   ; HL-Register mit &1234 laden.
30   PUSH HL          ; &1234 auf dem Stapel ablegen.
40   LD    DE,#5678   ; DE-Register mit &5678 laden.
50   PUSH DE          ; &5678 auf dem Stapel ablegen.
60   POP   BC         ; &5678 vom Stapel holen und in's
                      ; BC-Register laden.
70   POP   DE         ; &1234 vom Stapel holen und in's
                      ; DE-Register laden.
80   POP   HL          ; Unterschreitung des Stapelbodens.
                      ; Wert in HL unbekannt.
```

Adresse	LD SP,#C000	PUSH HL	PUSH DE	POP BC	POP DE	POP HL
&C002 →	&??	&??	&??	&??	&?? SP →	&??
&C001 →	&??	&??	&??	&??	&??	&??
&C000 → SP →	&??	&??	&??	&?? SP →	&??	&??
&BFFF →	&??	&12	&12	&12	&??	&??
&BFFE →	&?? SP →	&34	&34 SP →	&34	&??	&??
&BFFD →	&??	&??	&56	&??	&??	&??
&BFFC →	&??	&?? SP →	&78	&??	&??	&??
&BFFB →	&??	&??	&??	&??	&??	&??



Mit einem FLR-Array kann man auch sehr leicht einen Software-Stack realisieren. Die folgenden Assembler-Programme zeigen eine mögliche Variante. Dieser Stack wächst nach oben, und der Stackpointer zeigt jeweils auf das erste freie Byte über der Stapelspitze:

```
STACKP: DEFW #0000              ; Speicher für den Software-
                                ; Stackpointer
;
; PUSH    Eingabe: DE=Wert
;
STORE:  LD    HL,(STACKP)       ; Stackpointer holen
        LD    (HL),E            ;
        INC   HL                ; DE 'pushen':   LD (HL),DE mit
                                ; postinkrement
```

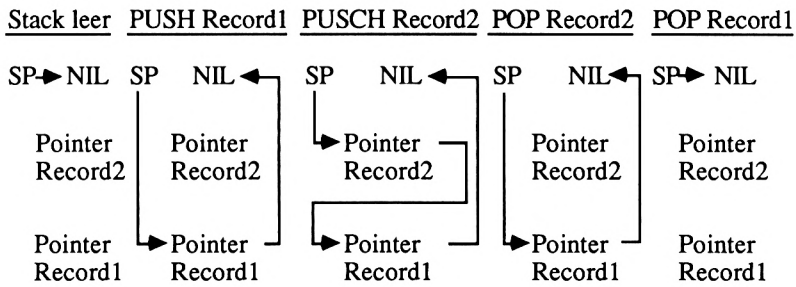
```
        LD    (HL),D          ;
        INC   HL              ;
        LD    (STACKP),HL     ; neuen Wert für den Stackp.
                              ; wieder speichern
        RET
;
; POP     Ausgabe: DE=Wert
;
RECALL: LD    HL,(STACKP)     ; Stackpointer holen
        DEC   HL              ;
        LD    D,(HL)          ; DE 'poppen':   LD DE,(HL) mit
                              ; predekrement
        DEC   HL              ;
        LD    E,(HL)          ;
        LD    (STACKP),HL     ; neuen Wert für den Stackp.
                              ; zurueckspeichern
        RET
```

Ein Stack als verkettete Liste

Verkettete Listen sind für Stacks eigentlich besonders gut geeignet, weil ja immer nur auf den ersten Record zugegriffen wird. Außerdem beanspruchen leere Stacks mit dieser Methode keinen Speicherplatz. Ein neuer Datensatz wird auf dem Stack abgelegt, indem er an erster Position abgelegt wird. Um ihn wieder herunterzunehmen, muß man ihn aus der Kette wieder ausklinken.

Solche Chains haben dabei sogar den Vorteil, daß der Stapelboden immer markiert ist, weil man in den Pointer des letzten Records des Stapels ja NIL eintragen kann. Ein maximal verfügbarer Bereich kann auch nicht überschritten werden, weil die Records in keinen reservierten Bereich kopiert werden, sondern an dem Platz belassen werden, an dem sie bereits stehen. Nur ihre Pointer werden verändert.

Die folgende Grafik demonstriert das "Pushen" und "Poppen" in einer "Linked List":



Eine mögliche Realisierung eines Linked-List-Stapels in Maschinencode zeigt das folgende Programm. Es ist dabei völlig uninteressant, wie lang die einzelnen Records sind und was sie enthalten. Wichtig ist nur, daß sie alle mit einem Pointer anfangen:

```
STACKP: DEFW #0000          ; Speicher für den Stackpointer
;
; PUSH: Eingabe: HL zeigt auf das LSB des Pointers des zu
; pushenden Records.
; ------------------------------------------------------------
STORE:  LD    DE,(STACKP)   ; Zeiger auf bisherigen ersten
                            ; Record nach DE holen.
;
        LD    (STACKP),HL   ; Zeiger auf neuen ersten Record
                            ; abspeichern.
;
        LD    (HL),E        ; Adresse des bisherigen ersten
                            ; und nun zweiten
        INC   HL            ; Records in den Pointer des neuen
                            ; ersten Records
        LD    (HL),D        ; eintragen.
        RET
;
;
; POP: Ausgabe: HL zeigt auf das LSB des Pointers des
; gepoppten Records.
;------------------------------------------------------------
RECALL: LD    HL,(STACKP)   ; Zeiger auf bisherigen ersten
                            ; Pointer nach HL holen.
;
        LD    E,(HL)        ; Zeiger auf bisherigen zweiten
                            ; und nun ersten
        INC   HL            ; Pointer nach DE holen.
        LD    D,(HL)
        DEC   HL            ; HL aber nicht veraendern!
;
        LD    (STACKP),DE   ; Bisherigen zweiten Pointer als
                            ; ersten eintragen.
        RET
```

QUEUES: FIRST IN – FIRST OUT

Gerade umgekehrt wie Stacks funktionieren Queues (Warteschlangen). Zwar kann man auch hier nur an einem Ende des "Stapels" Daten auflegen und an einem Ende Daten abholen, nur ist das Ende mit der Datenausgabe jetzt auf der anderen Seite des Stapels. Man legt also oben auf und holt unten ab.

Damit hat man beim Auslesen nicht den Zugriff auf das jeweils zuletzt auf den Stapel gelegte Datenelement, sondern auf das erste. Daraus resultiert auch die andere, für Queues gebrauchte Bezeichnung, FIFO-Speicher: First In – First Out.

Wie beim Stack benötigt man zwei Markierungen für das vordere und das hintere Ende. Außerdem ist der maximale Platzbedarf für solche Puffer, wenn sie als Array realisiert werden, nicht immer sicher zu bestimmen. Meist kann er sogar unendlich groß sein: beispielsweise bei Druckerpuffern, in denen Zeichen, die der Computer ausdrucken will, so lange zwischengespeichert werden, bis der Drucker sie abarbeiten kann. Denn der Computer kann normalerweise mit fast unbeschränkter Geschwindigkeit Zeichen abschicken; der Drukker aber ist immer bedeutend langsamer.

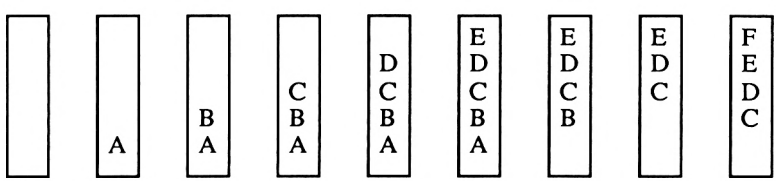
Im Gegensatz zu Stacks, wo meist derjenige, der einen Wert auf dem Stapel ablegt, diesen zu einem späteren Zeitpunkt auch wieder abholt, sind an Queues (Warteschlangen!) meist zwei Partner beteiligt: einer, der nachschiebt (Computer), und einer, der abholt (Drucker).

Während man bei Stacks also weiß, ob noch Platz da ist, muß man bei der Queue erst einmal nachschauen, ob der Puffer nicht vielleicht voll ist, und dann warten. Ebenso ist es aber möglich, daß der Abholer einmal schneller ist. Während man beim Stack wieder ganz genau weiß, daß noch ein Wert auf dem Stapel ist, muß man bei Queues erst einmal nachschauen und gegebenfalls warten.

Queues als FLR-Array

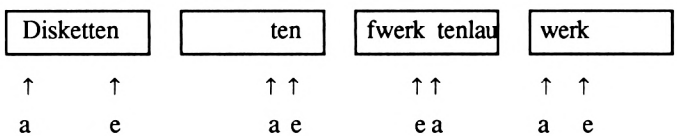
Auch hier sind die verketteten Listen im Vorteil, weil das Einfügen und Entfernen von Records vom Prinzip her besonders einfach ist. Bei Arrays müßte man jedesmal, wenn ein neues Element angehängt werden soll, die komplette Warteschlange um eine Position verschieben.

Im folgenden Beispiel wird der 5 Zeichen langen Puffer in Array-Form zunächst mit A bis E gefüllt; dann werden zwei Zeichen abgeholt und ein weiteres Zeichen nachgeschoben:



Geht man nicht sowieso davon aus, daß der Queue-Inhalt immer an eine Puffergrenze angerückt wird, also daß im oberen Beispiel alle Zeichen soweit nach unten nachrutschen, bis sie alle unten aufsitzen, so muß man doch spätestens im letzten Fall den Queue-Inhalt im Puffer verschieben. Das ist aber sehr zeitintensiv.

Diesen prinzipiellen Nachteil der Arrays bei der Bildung von Queues kann man aber umgehen, wenn man einen ringförmigen Speicher benutzt. Da man im Computer aber meist beschränkte lineare Adreßräume hat, muß man diese mit zusätzlichem Logikaufwand simulieren. Überschreitet einer der beiden Zeiger, die auf Ende und Anfang der Warteschlange zeigen, den zur Verfügung gestellten Bereich, so macht er einfach am anderen Ende des Puffers weiter:



Die folgenden Assembler-Routinen realisieren einen Wartespeicher für ASCII-Zeichen (oder Bytes; also FLR) in einem Array, der als Ringspeicher organisiert ist. Zeichen werden immer auf der Position von QUEEND (Schlangenende) angefügt. Der Schlangenkopf wird durch QUEANF angezeigt.

Der zusätzliche Aufwand beim Weiterstellen eines Zeigers ist im Unterprogramm VOR zusammengefaßt.

Die PUT- und GET-Routinen müssen zuerst testen, ob der Speicher voll bzw. leer ist. In diesem Fall können sie nur unverrichteter Dinge wieder zurückkehren. Das Problem ist aber zu erkennen, ob der Puffer nun voll oder leer ist:

Ist kein einziges Zeichen im Puffer, haben QUEANF und QUEEND denselben Wert. Wird der Puffer aber vollständig gefüllt, wird QUEEND genau einmal im Kreis weitergestellt und steht wieder auf seiner alten Position. Mithin läßt sich der Zustand "ganz voll" und "ganz leer" nicht so einfach unterscheiden.

Als Ausweg bietet sich an, den Puffer nie ganz zu füllen. Mindestens ein Zeichen muß frei bleiben. Der Zustand "Puffer voll" wird also per Definition bereits erreicht, wenn noch ein Byte im Puffer unbenutzt ist.

Der Test auf "Puffer voll" läßt sich also realisieren, indem QUEEND testweise erhöht und dann mit QUEANF auf Gleichheit verglichen wird. Ein leerer Puffer ist daran erkennbar, daß diese beiden Zeiger gleich sind, bevor man QUEEND erhöht.

```
BUFANF: DEFW #0000          ; Speicher für Pufferstartadresse
BUFEND: DEFW #0000          ; Speicher für Pufferende + 1
                            ; (Zeiger hinter Puffer)
;
QUEANF: DEFW #0000          ; Speicher fuer Zeiger auf erstes
                            ; Zeichen der Queue
QUEEND: DEFW #0000          ; Speicher für Zeiger auf
                            ; Queue-Ende + 1
;                           ; (Zeiger auf ersten freien Platz)
;
; Unterprog.: Stelle DE innerhalb des Puffers weiter
;
VOR:    INC  DE             ; stelle DE weiter
        LD   HL,(BUFEND)
        AND  A              ; CY := 0
        SBC  HL,DE          ; Hat DE das Pufferende erreicht?
        RET  NZ             ; nein
        LD   DE,(BUFANF)    ; sonst DE auf Pufferanfang
                            ; einstellen
        RET
;
PUT:    LD   DE,(QUEEND)    ; Zeiger auf 1. freien Platz am Ende
;                           ; der Schlange
        LD   (DE),A         ; Zeichen abspeichern
        CALL VOR            ; Zeiger weiterstellen
        LD   HL,(QUEANF)
        SBC  HL,DE          ; mit Zeiger auf Schlangenkopf
;                           ; vergleichen
        RET  Z              ; Puffer voll! QUEEND nicht
;                           ; weiterstellen. Das Zeichen
;                           ; wurde in dem Byte abgespeichert,
;                           ; das immer frei
;                           ; bleiben muss. Das Zeichen geht
                            ; verloren. CY = 0.
        LD   (QUEEND),DE    ; o.k.: QUEEND weiterstellen.
        SCF                 ; Erfolg im CY-Flag anmerken.
        RET
;
GET     LD   DE,(QUEANF)    ; Zeiger auf Schlangenkopf
        LD   HL,(QUEEND)    ; Zeiger auf Schlangenende
        AND  A              ; CY := 0
        SBC  HL,DE          ; vergleichen
        RET  Z              ; Puffer leer. CY = 0.
        LD   A,(DE)         ; sonst Zeichen vom Schlangenkopf
;                           ; nach A laden
        CALL VOR            ; und den Schlangenkopf um eine
```

```
;                              ; Position
        LD    (QUEANF),DE      ; weiterstellen (nach hinten
;                              ; ruecken).
        SCF                    ; Erfolg im CY-Flag anmerken.
        RET
```

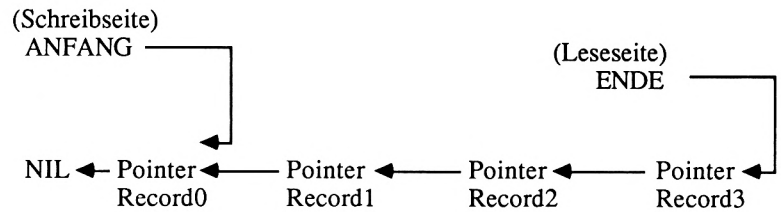
Eine solcher Ringpuffer ist im Tastatur-Manager enthalten und 20 Tastendrücke lang. Auf der einen Seite kann sich das laufende Programm die Zeichen abholen, auf der anderen Seite schiebt der Anwender durch seine Aktivitäten auf der Tastatur ständig Zeichen nach. Auch wenn das Programm kurzzeitig keine weiteren Zeichen abholen kann (weil es beispielsweise gerade einen besonders komplizierten Befehl abarbeitet), kann man noch eine Weile blind schreiben, ohne daß die Zeichen verlorengehen. Eine Warnung, wann der Puffer voll ist, erhält man allerdings nicht (manche Computer piepsen hier beispielsweise).

Ebenfalls nach dieser Methode sind die drei Puffer, die der Sound-Manager für jeden Tonkanal bereithält, aufgebaut. Diese sind jeweils vier Töne lang, wovon der erste Ton normalerweise gerade gespielt wird.

Queues als verkettete Liste

Verkettete Listen sind nicht nur besonders für Stacks, sondern auch für Warteschlangen geeignet, weil hier ebenfalls nur an je einem Ende gelesen und geschrieben wird. Da sich bei Queues aber an beiden Enden etwas tut, benötigt man zwei Pointer; einer zeigt auf den ersten und einer auf den letzten Record der Warteschlange.

Es wird aber trotzdem nur eine Hangelkette zwischen den einzelnen Records benötigt, die vom letzten, sprich ältesten Element, rückwärts bis zum Schlangenkopf führt:



Die im Kernel verwendete Warteschlange für die synchronous event pending queue ist eine verkettete Warteschlange. Sie ist aber etwas anders organisiert, weil hier die gekickten Event-Blocks entsprechend ihrer Priorität eingereiht werden.

Die folgenden Assembler-Routinen realisieren eine nach obigem Muster "verkabelte" Warteschlange. Wie beim Stack sind diese Routinen wieder universell sowohl für FL- als auch VL-Records anwendbar. Die Records werden ja nicht in einen reservierten Speicherbereich kopiert, sondern nur die Pointer verstellt.

Günstig bei Warteschlangen in "Kettentechnik" ist, daß man beim Nachschieben von Datensätzen nie abgewiesen werden kann, weil der Puffer voll ist. Diese Routine hängt ja nur einen bereits bestehenden Datenblock in die Kette ein. Es kann nur sein, daß irgendwann das gesamte System zusammenbricht, weil kein freier Speicher mehr vorhanden ist. Soll eine solche Warteschlange tatsächlich als Puffer zwischen zwei asynchron arbeitenden Einheiten dienen, ist es bestimmt sinnvoll, einen Zähler zu installieren, mit dem die in der Queue eingereihten Records auf eine bestimmte Höchstzahl begrenzt werden.

```
ANFANG: DEFW #0000          ; Zeiger auf ersten (neuesten)
;                           ; Record. Schreibseite.
ENDE:   DEFW #0000          ; Zeiger auf letzten (aeltesten)
;                           ; Record. Leseseite.

;
; Schiebe einen Record nach. Eingabe: HL zeigt auf LSB des
; Pointers.
;
PUT:    EX   DE,HL          ; Pointer auf neuen Record nach DE
;                           ; tauschen.
        LD   HL,(ANFANG)    ; Pointer auf den bisher letzten
;                           ; Record nach HL.
;
        LD   (HL),E         ; In den Pointer des bisher letzten
;                           ; Records
        INC  HL             ; die Adresse des neuen Records
;                           ; eintragen.
        LD   (HL),D         ;
;
        EX   DE,HL          ; Pointer auf neuen Record wieder
;                           ; nach HL tauschen
        LD   (ANFANG),HL    ; und in den Anfangszeiger
;                           ; eintragen.
;
PUT1:   LD   (HL),0         ; Pointer des neuen Records auf NIL
;                           ; einstellen.
        INC  HL             ;
```

```
        LD    (HL),0          ;
        RET
;
; Hole einen Record aus der Queue. Ausgabe: HL zeigt auf LSB
; des Pointers.
;
GET:    LD    HL,(ENDE)       ; HL := Pointer auf letzten Record.
        LD    A,H
        OR    L
        RET   Z               ; zurueck, wenn Queue leer.
;
        LD    E,(HL)          ; Pointer auf bisher vorletzten
;                             ; Record nach DE holen
        INC   HL              ;
        LD    D,(HL)          ;
        DEC   HL              ;
;
        LD    (ENDE),DE       ; und als neuen letzten Record
;                             ; eintragen
;
        LD    A,D             ; Queue jetzt leer?
        OR    E
        SCF                   ; Erfolg anmerken.
        RET   NZ              ; Nur zurueck, wenn die Queue jetzt
;                             ; nicht leer ist!
;
; Initialisieren
;
INITQU: LD    DE,ENDE         ; Adresse des Lesezeigers in den
;                             ; Schreibzeiger
        LD    (ANFANG),DE     ; eintragen (siehe Anmerkung).
;
        LD    DE,#0000        ; NIL in den Lesezeiger eintragen
        LD    (ENDE),DE
        RET                   ; Queue ist leer.
```

Anmerkung: Schwierig an der Behandlung der verketteten Queue ist, daß der Schreibpointer ANFANG immer auf den ersten, jüngsten Record zeigen muß. Ist die Queue aber leer, existiert dieser nicht.

Eine leere Warteschlange muß deshalb drei zusätzliche Bedingungen erfüllen: Erstens muß der Lesepointer ENDE auf NIL zeigen, damit kein weiterer Lesezugriff erfolgen kann. Zweitens darf ANFANG nicht einfach irgendwohin zeigen, da die hiermit adressierten Speicherzellen mit der Adresse des Records beschrieben werden, wenn der nächste (also erste) Record eingetragen wird. Drittens muß beim nächsten Nachschieben die Adresse des Records auch in den Lesepointer ENDE eingetragen werden, weil der Record als einziges Element in der Queue nicht nur der jüngste, sondern auch gleichzeitig der neue älteste wird. Die beiden letzten Fliegen werden mit einer Klappe geschlagen, und bei leerem Puffer wird der Schreibzeiger ANFANG auf ENDE eingestellt.

Datentypen

Das Betriebssystem des Schneider CPC kennt zwei unterschiedliche Zahlenformate: Integer und Real.

Mit *Integer* werden kleine, ganze Zahlen bezeichnet, die sich in 16 Bits, also einem Word abspeichern lassen.

Real sind Fließkommazahlen, deren interne Codierung 5 Bytes beansprucht. Hiermit können Zahlen auf etwa neun Stellen genau gespeichert werden. Die Darstellung umfaßt dabei den Bereich von etwa $\pm 10^{-38}$ bis $\pm 10^{+38}$. Für den alltäglichen Gebrauch ist das sicherlich ausreichend.

Die Z80-CPU selbst kennt mehrere Codierungsarten, die sich aber alle nur auf ganze Zahlen beziehen. Neben Bits sind Bytes und Words mit oder ohne Vorzeichen möglich und die Darstellung von Zahlen im sogenannten "Packed BCD"-Format wird unterstützt.

BCD-CODIERUNG

BCD heißt: "Binary Coded Decimals", also "binär codierte Dezimalzahlen". Die Dezimalziffern werden, meist in ihrer ASCII-Codierung, im Speicher nacheinander abgelegt. Zusätzlich werden meist noch zwei oder drei Bytes beansprucht, in denen Vorzeichen, Kommaposition und/oder ein Exponent gespeichert werden. Folgender Auszug stellt eine mögliche Codierung im BCD-Format für eine Zahl im Speicher dar:

```
ZAHL1:  DEFB VZE        ; Vorzeichen des Exponenten
        DEFB EXPONENT0  ; Exponent (1. Dezimalziffer)
        DEFB EXPONENT1  ; Exponent (2. Dezimalziffer)
        DEFB VZM        ; Vorzeichen der Mantisse
        DEFB ZIFFER0    ; erste Dezimalziffer (hoechstwertige)
        DEFB ZIFFER1    ;
        DEFB ZIFFER2    ;  .
        DEFB ZIFFER3    ;  .
        DEFB ZIFFER4    ;  .
        DEFB ZIFFER5    ;
        DEFB ZIFFER6    ; letzte Dezimalziffer (niederwertige)
```

Das reine BCD-Format wird vom Z80 nicht unterstützt und ist ansonsten auch recht unbeliebt, weil es zuviel Speicherplatz beansprucht. Sollen Zahlen in Strings oder in Dateien auf Massenspeichern gespeichert werden, kann ein solches Format aber sinnvoll sein, um zu vermeiden, daß einzelne Bytes innerhalb der Zahl zufällig ein Steuerzeichen darstellen.

PACKED BCD

Der Hauptnachteil des BCD-Formates, der übergroße Speicherplatzbedarf, wird bei der Verwendung von "Packed BCD" verringert. Zur eindeutigen Darstellung einer Dezimalziffer sind nämlich nur vier Bits notwendig. Hiermit lassen sich Zahlen von 0 bis 15 codieren, wobei für Dezimalziffern natürlich nur die Werte 0 bis 9 zulässig sind. In einem Byte lassen sich so zwei Dezimalzahlen unterbringen, wobei die eine im höherwertigen Nibble (Bits 4 bis 7) und die andere im niederwertigen Nibble (Bits 0 bis 3) zu liegen kommt.

Der Z80 unterstützt das Packed-BCD-Format mit seinem Befehl DAA. Hiermit kann nach jeder normalen binären Addition oder Subtraktion mit dem A-Register das Ergebnis entsprechend der BCD-Darstellung korrigiert werden. Aber auch die Befehle RLD (HL) und RRD (HL) unterstützen die nibbleweise Behandlung von BCD-Zahlen.

Das Betriebssystem des Schneider CPC benutzt dieses Format nie. Eine Zahl könnte im Packed-BCD-Format aber wie folgt im Speicher abgelegt sein:

```
ZAHL1:  DEFB VZE        ; Vorzeichen des Exponenten
        DEFB EXPONENT   ; Exponent, BCD-codiert
        DEFB VZM        ; Vorzeichen der Mantisse
        DEFB ZIFFERN01  ; 1. & 2. Dezimalziffern
                        ; (hoechstwertig)
        DEFB ZIFFERN23  ; .
        DEFB ZIFFERN45  ; .
        DEFB ZIFFERN67  ; .
        DEFB ZIFFERN89  ; 9. & 10. Dezimalziffer (niederwertig)
```

BYTES

Der Z80 unterstützt als 8-Bit-CPU in der Hauptsache das Rechnen mit Bytes. Das Haupt-Rechenregister ist der Akku, der für Addition und Subtraktion immer benötigt wird. Einfaches Inkrementieren und Dekrementieren ist aber mit allen anderen Registern auch möglich.

Im Schneider CPC werden Bytes direkt nur für die Speicherung von (ASCII-) Zeichen benutzt, beispielsweise in Strings. Mithin besteht in BASIC keine vorgegebene Möglichkeit, mit Bytes zu rechnen oder Byte-Variablen anzulegen, was beispielsweise bei großen Datenfeldern von Nutzen sein kann.

Bytes können von der CPU sowohl vorzeichenbehaftet als auch nur positiv behandelt werden. Im ersten Fall können Zahlen von –128 bis +127 und im zweiten Fall von 0 bis 255 dargestellt werden.

Negative Zahlen werden in komplementärer Form dargestellt. Die Zahlen von 0 bis 127 sind in beiden Fällen identisch. Die Zahlen –128 bis –1 entsprechen 128 bis 255. Ein gesetztes siebtes Bit signalisiert bei der Komplementärdarstellung eine negative Zahl.

Dabei gibt es keine Unterscheidungsmöglichkeit für die beiden Darstellungsarten. Ein Byte wird nur "per Definition" der einen oder anderen Art zugeordnet. Auch die Rechenoperationen sind identisch! Man muß nur, je nach Darstellungsart, andere Flags auswerten, um eine Überschreitung des darstellbaren Bereichs festzustellen.

Für die normale Darstellung nur positiver Zahlen muß für einen Übertrag das Carry-Flag getestet werden. Beim Rechnen mit Komplementärzahlen muß man das Parity/Overflow-Flag auswerten.

Die folgende Tabelle zeigt eine Gegenüberstellung der Interpretation eines Bytes (im Akku etwa) als Zahl mit oder ohne Vorzeichen:

```
Byte = kompl / pos
-----------------
 &02 =   2    /   2
 &01 =   1    /   1
 &00 =   0    /   0
 &FF =  -1    / 255
 &FE =  -2    / 254
    ...
 &81 = -127   / 129
 &80 = -128   / 128
 &7F =  127   / 127
 &7E =  126   / 126
```

WORDS – INTEGER

Obwohl der Z80 nur ein 8-Bit-Mikroprozessor ist, enthält sein Befehlssatz bereits Kommandos, um mit 16 Bit breiten "Words" zu rechnen. Als Hauptrechenregister dient hierbei HL. Alle 16-Bit-Doppelregister können eine nur positive Zahl im Bereich von 0 bis 65535 oder eine vorzeichenbehaftete Zahl zwischen –32768 und +32767 enthalten. Für letztere wird wieder die Komplementärdarstellung benutzt, die wie bei den Bytes gehandhabt wird.

Die Darstellung der Integer-Zahlen in BASIC und Betriebssystem entspricht der Codierung im Z80, wobei natürlich Zahlen mit Vorzeichen benutzt werden. Eine weitere Vorgabe des Prozessors ist die Reihenfolge, mit der die beiden Bytes eines Integer-Words im Speicher stehen: Auf der niedrigeren Adresse steht das niederwertige Byte und auf der folgenden, höheren Adresse das höherwertige Byte mit dem Vorzeichen.

Die folgende Tabelle zeigt eine Gegenüberstellung der Interpretation eines Words (in HL etwa) als Zahl mit oder ohne Vorzeichen:

```
 Word = kompl   /    pos
-------------------------
&0002 =      2  /      2
&0001 =      1  /      1
&0000 =      0  /      0
&FFFF =     -1  / 65535
&FFFE =     -2  / 65534
     ...
&8002 = -32766 / 32770
&8001 = -32767 / 32769
&8000 = -32768 / 32768
&7FFF =  32767 / 32767
&7FFE =  32766 / 32766
```

REAL

Grundsätzlich werden Fließkommazahlen mit 2 Komponenten dargestellt: durch eine *Mantisse* und einen *Exponenten*. Die Mantisse gibt die Ziffernfolge an und der Exponent die Lage des Kommas. Der Wert einer solchen Zahl ergibt sich dann aus:

mantisse * z ^{Exponent}

wobei z die Zahlenbasis ist: 10 bei dezimaler oder 2 bei binärer Darstellung. Als Beispiel eine Zahl in Exponentialschreibweise im Dezimalsystem:

Mantisse = 12,3456
Exponent = 3 → $12{,}3456 * 10^3 = 12345{,}6$

Die Fließkomma-Rechenroutinen des Schneider CPC benutzen die Exponentialform, um Real-Zahlen zu codieren. Sinnvollerweise rechnet der CPC aber im Binärsystem. Die gesamte Zahl beansprucht dabei 5 Bytes. Die ersten vier Bytes enthalten die Mantisse einschließlich Vorzeichen und das fünfte Byte den Binärexponenten:

```
ZAHL1:  DEFB M0
        DEFB M1
        DEFB M2
        DEFB M3 + VZ
        DEFB EXP+128
```

DIE MANTISSE

M0 bis M3 bilden die Mantisse, wobei M3 das höchstwertige Byte ist. Die Mantisse wird in normalisierter Form angegeben: 0,1xxxxxx... (binär!).

Dadurch kann die Mantisse Werte zwischen 0,5 (incl.) und 1,0 (excl.) annehmen:

[0,5 ... Mantisse ... [1,0

Normalisiert bedeutet, daß die signifikanten Ziffern der Mantisse quasi von hinten gegen den Dezimalpunkt gerückt werden. Vor dem Komma steht somit immer eine Null und dahinter eine Eins.

In M0 bis M3 wird nur der Nachkommateil der Mantisse gespeichert. Irgendeine Vornull mit abzuspeichern wäre nur unnötiger Ballast.

Außerdem steht aber auch die Eins nach dem Komma fest. Anders als im Dezimalsystem kann die erste Nachkommastelle der normalisierten Mantisse nur den Wert Eins annehmen. Sie soll ja ungleich Null sein. Und im Binärsystem bleibt da nur die Eins als einzige Alternative.

Auch diese Ziffer explizit abzuspeichern, ist überflüssig. Trotzdem wird diese Stelle in der Mantisse mitgeführt. Sie belegt Bit 7 von M3. Daß ihr Wert aber feststeht, hat man sich zunutze gemacht und hier einfach das Vorzeichen gespeichert. Ist dieses Bit gesetzt, ist die Zahl negativ. Erst zum Rechnen wird das Vorzeichen herausgezogen, getrennt gespeichert und diese Ziffer (Bit 7 von M3) auf Eins gesetzt.

DER EXPONENT

Um sehr kleine Zahlen darstellen zu können, muß die Darstellung des Exponenten auch negative Zahlen ermöglichen. Normalerweise bedient man sich dafür des Zweierkomplements. Beim Exponenten hat es sich aber eingebürgert, einen bestimmten, festen Betrag zu addieren. Dadurch erhält man die sogenannte "Charakteristik".

Den Offset wählt man normalerweise so, daß das Komma der Mantisse gleich weit nach links und nach rechts verschoben werden kann. Der Exponent "Null" muß also möglichst in der Mitte des für die Charakteristik zur Verfügung stehenden Zahlenbereichs liegen. Da der Exponent (bzw. die Charak-

teristik) ein Byte beanspruchen kann, ergibt sich als "Mittelwert" 255/2 = 127,5. Üblicherweise wird aufgerundet, weil der kleinste zur Verfügung stehende Wert für eine Sonderfunktion reserviert ist.

Auch die im Schneider CPC integrierte Fließkomma-Arithmetik bildet die Charakteristik, indem sie 128 zum Exponenten addiert. Der Exponent Null, was soviel bedeutet wie "Keine Verschiebung der Kommaposition in der Mantisse", entspricht der Charakteristik 128.

Der größtmögliche Exponent ist 255 – 128 = 127. Die Mantisse kann also maximal um 127 Binärstellen nach links verschoben werden, was einer Multiplikation mit 2^{127} entspricht. Umgerechnet ergibt das einem maximal möglichen Dezimalexponenten von etwa +38,2.

Der kleinstmögliche Exponent entspricht der Charakteristik 1 (0 hat eine Sonderbedeutung) und ist daher 1 – 128 = –127. Das entspricht einem Dezimalexponenten von –38,2. Die Mantisse kann also auch um maximal 127 Binärstellen nach rechts verschoben werden.

NULL

Die Konvention, daß die Mantisse in ihrer normalisierten Form vorliegen soll, bereitet aber einer Zahl enorme Schwierigkeiten: der Null.

Zahlen, deren Betrag größer als $(0{,}111111...)_2 * 2^{127}$ ist, lassen sich nicht mehr darstellen. Nach Dezimal konvertiert entspricht das etwa $1{,}70141*10^{38}$, was man leicht überprüfen kann, wenn man den Computer $10^{38}*9$ berechnen läßt. In diesem Fall wird ein "Overflow error" gemeldet:

```
PRINT 1e38*9 [ENTER]
Overflow
 1.70141E+38
Ready
```

Andererseits lassen sich aber auch keine Zahlen darstellen, deren Betrag kleiner als $(0{,}1000...)_2 * 2^{-127}$ ist! Die betragsmäßig kleinste Zahl, die der Schneider CPC darstellen kann, läßt sich ganz leicht ausrechnen:

```
PRINT 0.5*2^-127 [ENTER]          : REM (0,5)10 = (0,1)2
 2.93874E-39
Ready
```

Noch kleinere Zahlen werden immer auf Null gerundet. An dieser Stelle entsteht also ein Genauigkeitssprung: Solange eine Zahl nicht nach Null gerundet werden mußte, bleibt das Ergebnis einer Rechenoperation normalerweise auf etwa 9 Stellen genau. Die Rechnung a/b*b wird einen Wert liefern, der nicht oder nur kaum von a abweicht. Muß aber das Ergebnis des Terms a/b auf Null gerundet werden, wozu im Einzelfall eine ganz geringe Variation von a und b genügt, wird das Ergebnis von a/b*b ebenfalls Null sein.

Zur Darstellung der Zahl Null selbst wird ein Ausnahmefall in der Zahlencodierung konstruiert: Der allerkleinste darstellbare Exponent bleibt ausschließlich für die Null reserviert! Ist der Exponent –128, wird die Mantisse also nicht ausgewertet. Die Charakteristik ist dann 0, was sich besonders einfach testen läßt.

DEZIMALWANDLUNG

Eine Fließkommazahl, die in binärer Mantisse/Exponent-Schreibweise abgespeichert ist, ist nur mit großen Schwierigkeiten dezimal ausdruckbar! Das liegt daran, daß man nicht einfach die Mantisse und den Exponenten getrennt ins Dezimalsystem wandeln kann und so die Dezimalzahl erhält. Die Basis des Exponenten ist ja ebenfalls verschieden. Eine Erhöhung des Binärexponenten bedeutet eben "nur im Binärsystem" einfach eine Verschiebung der Kommaposition um eine Stelle. Im Dezimalsystem muß man sie zu Fuß als das bearbeiten, was es ist: eine Multiplikation mit Zwei.

Für den umgekehrten Weg, die Auswertung einer Ziffernfolge, die in dezimaler Exponentialschreibweise eingegeben wurde, um sie intern abzuspeichern, gilt dasselbe. Hier muß binär für jeden Dezimalexponenten die Mantisse mit 10 multipliziert werden.

Die Vorteile binärer Codierung liegen deshalb ausschließlich in ihrem geringeren Speicherplatzbedarf und in der höheren Rechengeschwindigkeit. Eine BCD-Codierung (auch Packed BCD) hat dagegen eindeutige Vorteile, wenn Zahlen oft ausgedruckt werden sollen und der Rechenaufwand in den Hintergrund tritt, also beispielsweise bei der Darstellung von Tabellen und Dateien.

Da der Schneider CPC in erster Linie ein "Rechner" ist, und in akzeptabler Zeit auch einen komplizierteren trigonometrischen Funktionenplot auf dem Bildschirm darstellen soll, ist die Entscheidung bei AMSTRADd für die binäre Codierung sicher nur zu begrüßen.

STRINGS

Neben Integer- und Fließkomma-Zahlen kennt der BASIC-Interpreter des Schneider CPC noch einen weiteren Datentyp: Zeichenketten (Strings).

Die Codierung von Strings ist von der CPU nur sehr schwach vorgegeben. Allenfalls die Tatsache, daß der CPC für jeden Buchstaben ein Byte verwendet, wird von ihr bestimmt.

Sind die Integer- und Fließkomma-Rechenroutinen nur mit Vorsicht dem Betriebssystem zuzurechnen, so ist für die Bearbeitung von Zeichenketten ausschließlich der BASIC-Interpreter selbst zuständig.

Eine prinzipielle Schwierigkeit bei der Speicherung von Strings liegt darin, daß sie mit jeder "Rechenoperation" ihre Länge ändern können. Sie können beim CPC deshalb nicht direkt im Variablenbereich des BASIC-Interpreters untergebracht werden. Dieser verfügt ja über precompilierende Fähigkeiten, für die im Variablenbereich feste Adressen benötigt werden. Strings dürfen sich hier also nicht ausdehnen und wieder zusammenziehen, wodurch sich die Lage aller dahinter abgelegten Variablen verschieben würde.

Das von AMSTRAD hier verwendete Verfahren ist aber recht verbreitet. Zur Speicherung eines Strings wird im Variablenbereich nur ein sogenannter "Stringdescriptor", ein String-Beschreiber eingetragen. Dieser ist immer drei Bytes lang, egal ob der String selbst aus 0, 1, 2 oder 255 Zeichen besteht.

Der Stringdescriptor ist wie folgt aufgebaut:

```
DESCR1: DEFB LAENGE
        DEFW ADRESSE
```

Das erste Byte gibt an, wie lang die Zeichenkette ist. Dadurch ergibt sich die Beschränkung auf maximal 255 Zeichen. Die beiden folgenden Bytes bilden einen Zeiger auf den String, der bei einer Länge "Null" natürlich bedeutungslos ist.

Der String kann dabei entweder im Programmtext selbst liegen oder im "Memory Pool", also dem frei verfügbaren Speicherbereich. Bei einer einfachen Zuweisung eines Textes zu einer String-Variablen innerhalb eines Programms wird der String nicht in den Memory Pool kopiert, sondern der Zeiger einfach auf die Zeichenfolge im BASIC-Programm eingestellt. Das ist wichtig zu beachten, wenn ein Maschinencode-Programm einen String ändert, damit nicht im Programmtext selbst Änderungen vorgenommen werden!

Sobald einer String-Variablen eine neue Zeichenkette zugeordnet wird (durch Verknüpfung mit einem anderen String oder die Anwendung von MID$, LEFT$, RIGHT$), wird diese im Memory Pool angelegt. Das folgende Beispiel zeigt das:

```
10 LET a$="123456" : REM Descr. von a$ zeigt in das BASIC-
                         Programm hinein
20 LET a$=a$+"789" : REM a$ wird im Memory Pool neu
                         eingerichtet
30 LET b$="1234"+"": REM BASIC muss rechnen, deshalb wird b$
                         auch in den Memory Pool gelegt; guter
                         Trick, wenn man jetzt b$ an ein
                         Maschinenprogramm uebergeben will,
                         das b$ aendert.
40 CALL &AF00,@b$  : REM zum Beispiel an dieses.
```

GARBAGE COLLECTION

Jede Manipulation an einem String bewirkt, daß die alte Zeichenkette einfach vergessen wird und die neu gebildete im noch freien Speicherbereich angelegt wird. Dieser wird dadurch natürlich immer kleiner, und irgendwann ist er vollständig aufgebraucht. Dann kommt es zur "Garbage Collection", der Müllsammlung, bei der alle unbenutzten Speicherstellen ermittelt und zu einem neuen Memory Pool zusammengezogen werden.

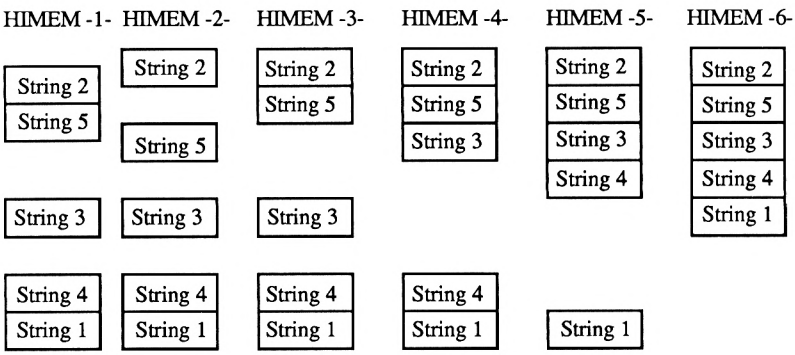
Dabei wurde ab dem CPC 664 eine entscheidende Änderung bei der Abspeicherung von Strings vorgenommen, die vor allem die Garbage Collection in ausgedehnten String-Feldern beschleunigt.

... beim CPC 464

Beim CPC 464 werden die Strings folgendermaßen gespeichert: Im Variablenbereich steht zu jedem String ein Descriptor. Dieser zeigt auf einen String, der entweder im BASIC-Programm selbst (s.o.) oder im Memory Pool liegt. Im Memory Pool liegen alle Strings dicht an dicht. Die einzelnen Zeichenketten können nur mittels der Descriptoren auseinandergehalten werden. Ab und zu liegt dazwischen ein vergessener String, zu dem kein Descriptor mehr existiert. Je weiter es auf die nächste Garbage Collection zugeht, desto mehr Lücken gibt es.

Um nun die Lücken zu schließen und wieder dem freien Speicherbereich zuzuordnen, geht der BASIC-Interpreter wie folgt vor: Alle Descriptoren werden durchsucht, um den am höchsten im Memory Pool gelegenen String zu bestimmen. Dieser wird jetzt nach oben an HIMEM angerückt und die Adresse

im Descriptor aktualisiert. Dann werden alle Descriptoren erneut durchsucht, um den nächsthöchsten String zu finden. Dieser wird dann ebenfalls bündig unten an die oberste Zeichenkette "rangeschoben" und der Descriptor aktualisiert. Das geht so weiter, bis kein Descriptor mehr auffindbar ist oder nur noch Descriptoren vorhanden sind, deren Zeiger in den Programmtext zeigen. Die folgende Grafik veranschaulicht den Prozeß:



Enthält der Variablenbereich n String-Descriptoren, so müssen n Zeichenketten verschoben werden. Schlimmer ist jedoch, daß für jede der n Zeichenketten alle n Descriptoren abgesucht werden müssen, um den noch verbliebenen höchsten String zu finden. Es sind deshalb n * n Vergleiche von String-Adressen nötig; der Aufwand steigt *quadratisch* mit der Anzahl der definierten Strings!

Vor allem bei umfangreichen String-Feldern (z.B.: DIM a$(499,1) = 1000 Strings = 1 Million Vergleiche!) benötigt BASIC für eine Garbage Collection einige Zeit.

... beim CPC 664 und 6128

Speziell für professionelle Anwendungen wird das ansonsten sehr schnelle Locomotive BASIC hier inakzeptabel langsam.

In der Version 1.1 des BASIC-Interpreters hat man deshalb das Abspeicherungsformat der Zeichenketten im Memory Pool geändert. Zusätzlich zu jedem String werden noch zwei Bytes reserviert, in denen normalerweise nur die String-Länge abgespeichert wird (zusätzlich zur Längenangabe im De-

scriptor). Durch diese zwei Bytes wird der Platzverbrauch pro String erhöht. Mithin kann es vorkommen, daß BASIC-Programme, die auf dem CPC 464 gerade noch laufen, auf einem CPC 664 oder 6128 mit der Fehlermeldung

```
String space full
```

passen müssen. Entscheidende Vorteile hat dieses Verfahren aber, sobald eine Garbage Collection durchgeführt werden muß. Hier geht der BASIC-Interpreter jetzt nämlich vollkommen anders vor.

Zunächst werden alle Descriptoren im Variablenbereich abgefragt, und in die zwei Bytes vor jedem String, der im Memory Pool liegt, die Adresse des Descriptors eingetragen. Danach zeigt also in jedem Descriptor ein Zeiger auf den String und in jedem (noch benutzten!) String ein Zeiger zurück auf den Descriptor.

Im zweiten Durchgang fängt BASIC mit dem untersten String an, dessen Lage bekannt ist, weil hierauf eine Systemvariable zeigt. Die Lage des untersten Strings im Memory Pool muß ja jederzeit bestimmbar sein, um hier weitere Zeichenketten anzufügen.

Hier können nun zwei Fälle auftreten. Ist die "Adresse", die in den zwei Bytes zu diesem String eingetragen wurde, kleiner oder gleich 255, so handelt es sich wohl nicht um die Adresse des zugehörigen Descriptors, sondern um die String-Länge. Die Länge wurde im ersten Durchgang nicht gegen die Descriptor-Adresse ausgetauscht, also gibt es keinen Descriptor mehr zu dieser Zeichenkette. Es handelt sich also um einen der vielen "vergessenen" Strings, die nun aus dem Speicher gelöscht werden können. Mit Hilfe der Länge kann sofort der Anfang des nächsten Strings festgestellt werden.

Der andere Fall ist, daß tatsächlich eine Adresse gefunden wird. Dann wird der String bündig (plus zwei Bytes) an den letzten String nach unten angerückt und der Zeiger im Descriptor aktualisiert. Mit Hilfe der alten Lage des Strings und der Länge (die nun aus dem Descriptor bestimmt werden muß) läßt sich auch hier der nächste String finden.

Das geht so lange weiter, bis HIMEM erreicht ist. Danach wird in einem dritten Durchgang das gesamte String-Paket von der Unterkante des Memory Pools bündig nach oben an HIMEM herangeschoben und in keinem vierten Durchgang alle Descriptoren um die Verschiebeweite erhöht.

Das klingt nicht nur umständlicher, das ist es auch. Aber dieses Verfahren hat einen entscheidenden Vorteil: Die Dauer einer Garbage Collection nimmt jetzt nicht mehr quadratisch mit der Anzahl der definierten String-Variablen zu!

Das folgende Testprogramm wurde auf einem CPC 464 und auf einem CPC 6128 getestet:

```
100 DIM a$(0):z=TIME:a=5
110 WHILE a<1000
120 a=a+a
130 ERASE a$
140 DIM a$(a)
150 FOR i=0 TO a:a$(i)="#"+"":NEXT
160 t=TIME:a$(0)=SPACE$(255)
170 IF t+5>TIME THEN 160
180 PRINT CHR$(7)
190 WEND
```

Zwar war ursprünglich gedacht, die verbrauchten Zeiten mit der Systemvariablen TIME auf 1/300 Sekunden genau zu messen. Das ist aber leider nicht möglich, weil während einer Garbage Collection zeitweise der Interrupt abgestellt wird. Die Zeiten sind also mit der Hand gestoppt, sprechen aber für sich.

Das Programm dimensioniert in der WHILE/WEND-Schleife immer größere String-Arrays (Zeilen 120-140), die dann in Zeile 150 "gefüllt" werden, damit es auch wirklich etwas im Memory Pool zu verschieben gibt. Danach wird in Zeile 160/170 so lange a$(0) verändert, bis eine Garbage Collection notwendig wird. Das kann man (trotz abgestelltem Interrupt) noch gut an der Systemvariablen TIME erkennen.

Folgende Zeiten wurden gemessen:

	CPC 464		CPC 6128	
DIM	Zeit ab Start	/ Differenz	Zeit ab Start	/ Differenz
a$(10)	001.5 Sek.	001.5 Sek.	001.5 Sek.	001.5 Sek.
a$(20)	003 Sek.	001.5 Sek.	003 Sek.	001.5 Sek.
a$(40)	005.5 Sek.	002.5 Sek.	004.5 Sek.	001.5 Sek.
a$(80)	008 Sek.	002.5 Sek.	006 Sek.	001.5 Sek.
a$(160)	013 Sek.	005 Sek.	008 Sek.	002 Sek.
a$(320)	028 Sek.	015 Sek.	011 Sek.	003 Sek.
a$(640)	078 Sek.	050 Sek.	015 Sek.	004 Sek.
a$(1280)	266 Sek.	188 Sek.	021 Sek.	006 Sek.

Programmiertechnik

PFLICHTENLISTE

Zunächst muß man sich überhaupt einmal klar werden, was das Programm machen, bzw. können soll. Eine Pflichtenliste, schriftlich abgefaßt, ist ratsam. Dabei sollte man die einzelnen Anforderungen schon etwas systematisieren. Eine Pflichtenliste ist dabei nicht so endgültig, wie man vielleicht denken könnte.

GLIEDERUNG

Man zergliedert am besten das Gesamtpaket in einzelne Unterprobleme, die, hätte man sie alle schon gelöst, das gesamte Programm enthalten würden. Diese Unterprobleme lassen sich natürlich wieder untergliedern, die so erhaltenen Unter-Unterprobleme auch usw.

Dabei darf der Zusammenhalt zwischen den einzelnen Teilproblemen nicht verlorengehen. Es empfiehlt sich also auch hier, seine Gedanken in irgendeiner schriftlichen Form festzuhalten. Ob man auf eine grafische Methode zurückgreift (Flußdiagramme o.ä.) oder das Ganze in verständliche, leserliche Worte faßt, ist dabei weitgehend egal.

PROGRAMMBIBLIOTHEKEN

Viele Programmierer stellen sich ganze Programmbibliotheken mit fertigen Unterprogrammen für die verschiedensten Probleme zusammen. Mit jedem neu geschriebenen Programm wächst mit Sicherheit auch die Bibliothek.

MODULARISIERUNG

Am wichtigsten von allen Methoden ist aber die klare Gliederung eines Problems. Umfangreiche Probleme lassen sich nur bewältigen, indem man sie zergliedert und die einzelnen Probleme getrennt löst.

Um den Überblick zu behalten, muß man die einzelnen Bruchstücke (ob nun schon gelöst oder nicht) als "black box" betrachten. Nicht, wie ein Unterprogramm die ihm übertragenen Funktionen ausführt, nur das, was nach außen sichtbar wird, ist interessant. Damit wird aus einem Unterprogramm ein "Modul".

Die Vektoren des Betriebssystems im RAM stellen in diesem Sinne über 200 Programm-Module bereit. Genau definierte Probleme sind im Betriebssystem gelöst und können über die Vektoren aufgerufen werden. Für den Programmierer ist es völlig unwichtig zu verstehen, wie man die einzelnen Probleme bei AMSTRAD gelöst hat. Ihn interessiert nur die Schnittstellenbeschreibung: Eingaben, Ausgaben und Wirkung der einzelnen Routinen.

KOMMENTARE

Nicht zu unterschätzen ist eine gute Dokumentation. Je niedriger die Programmiersprache gewählt ist, umso höher wird der Anteil der "Remarks". Während der Programmentwicklung sollte man seinen Programmtext mit zahlreichen Kommentaren versehen, die man in der Arbeitsversion wieder herausnehmen kann.

Programmstrukturen

So verschieden die einzelnen Programmiersprachen auch sind, so gibt es doch einige grundlegende Strukturen, die fast allen gemein sind. Dabei kann man den Befehlsumfang eines Interpreters, Compilers oder Assemblers grob in drei Gruppen einteilen:

- Aktionen
- Strukturen
- Steueranweisungen

Die "Aktionen" umfassen alle Befehle, die "etwas bewirken" wie etwa PRINT, PLOT oder Wertzuweisungen wie LET.

"Steueranweisungen" sind Befehle, die normalerweise die Programmausführung nicht beeinflussen, wie beispielsweise TRON und TROFF zur Fehlersuche in BASIC oder ORG, der Befehl, der in fast allen Assemblern die Startadresse für den zu generierenden Maschinencode festlegt.

Unter dem Begriff "Programmstrukturen" werden alle Befehle zusammengefaßt, die den Ablauf eines Programms verändern. Ein Programm, das nur aus strukturierenden Befehlen besteht, würde nichts bewirken. Die Strukturen verändern aber den linearen Ablauf des Programms und bestimmen so die Reihenfolge, in der die Aktionen abgearbeitet werden. Beispiele sind GOTO, GOSUB, WHILE-WEND usw.

VERZWEIGUNGEN – DER EINFACHE SPRUNG

Die einfachste Struktur überhaupt, die fast jede Sprache bereithält, ist der Sprung.

Normalerweise wird der Programmtext Befehl für Befehl abgearbeitet. Dafür benutzt die Befehlsholeinheit des Mikroprozessors oder des BASIC-Interpreters einen Zeiger. Der Zeiger zeigt auf einen Befehl, der eingelesen wird, und der Zeiger wird weitergestellt. Dann wird der Befehl bearbeitet. Dieser Zyklus läuft, mit geringen Variationen, immer wieder gleich ab:

1. angezeigten Befehl einlesen
2. Zeiger weiterstellen
3. Befehl ausführen

Beim Sprungbefehl wird der Befehlszeiger in der Befehlsausführphase auf die angegebene Stelle eingestellt. Er wird mit einer neuen Adresse geladen. In BASIC muß man dafür Zeilennummern angeben, in Assembler Speicheradressen, die aber fast ausschließlich mit symbolischen Namen, den sogenannten Labels, bezeichnet werden:

BASIC	Assembler
50 GOTO 100	LABEL1: JP LABEL2

Mit solchen unbedingten Sprüngen (unbedingt – ohne Bedingung) kann man aber nur endlose Schleifen bilden (indem man immer wieder zurückspringt) oder zwei Programmpfade zusammenführen:

Endlos-Schleife

```
50 PRINT "#";
60 GOTO 50
```

Zusammenführen von Programmpfaden:

```
50 LET C=100+250 : REM Berechnung 1
60 GOTO 80
70 LET C=125+175 : REM Berechnung 2
75 '
80 PRINT C
85 REM Gemeinsame Druckroutine
90 END
```

Man kann eine Verzweigung aber auch an eine Bedingung knüpfen und spricht dann vom bedingten Sprung. Hiermit sind prinzipiell bereits alle Probleme

lösbar. Beispielsweise kann man so Schleifen mit Abbruchkriterium herstellen:

```
100 LET I=1
110 PRINT I
120 LET I=I+1
130 IF I<25 THEN GOTO 110
140 STOP
```

```
ZAEHLE: LD   A,1
Z1:     PUSH AF
        CALL PRINT
        POP  AF
        INC  A
        CP   25
        JR   C,Z1
        RET
```

BEDINGTE BEARBEITUNG VON BEFEHLEN

Die einfachen Sprungbefehle sollten nach Möglichkeit völlig vermieden werden, da sie ein Programm unübersichtlich machen. Jede höhere Sprache stellt deshalb Strukturierungsmittel bereit, die sich zwar alle auf bedingte Sprünge zurückführen lassen, bei denen aber Sinn und Zweck der Verzweigungen offensichtlicher wird.

BASIC kennt sogar überhaupt keinen echten bedingten Sprung. Im obigen Beispiel wurde ein solcher nur mit Hilfe eines BASIC-typischen Strukturierungsmittels simuliert. Mit den Befehlen IF, THEN und ELSE lassen sich Programmteile nur dann bearbeiten, wenn eine bestimmte Bedingung erfüllt ist:

```
 4 REM HI-LO-Spiel
 5 REM -----------
10 z=INT(RND*1000)
20 INPUT "rate:",e
30 IF e=z THEN PRINT "richtig.":GOTO 10
40 IF e<z THEN PRINT "zu klein."
          ELSE PRINT "zu gross."
50 GOTO 20
```

In HI-LO gilt es, eine Zahl z zu erraten. Nach der Eingabe von e muß das Programm drei Fälle unterscheiden. In Zeile 30 wird getestet, ob die Zahl erraten wurde. Nur wenn e gleich z ist, wird der Programmteil nach THEN abgearbeitet. Dieser besteht aus zwei Befehlen.

Ist e=z verzweigt das Programm zurück nach Zeile 10 (endlose Schleife). Wenn nicht, wird Zeile 40 abgearbeitet. Hier wird untersucht, ob e kleiner als z ist. Ist das der Fall, wird im THEN-Pfad die Meldung "zu klein" ausgegeben. Wenn nicht, werden die Befehle ab ELSE abgearbeitet.

In BASIC können Fallabfragen auch geschachtelt werden. Das vorherige Beispiel läßt sich dann wie folgt verkürzen:

```
 4 REM HI-LO-Spiel
 5 REM -----------
10 z=INT(RND*1000)
20 INPUT "rate:",e
30 IF e=z THEN PRINT "richtig.":GOTO 10
         ELSE IF e<z THEN PRINT "zu klein."
                     ELSE PRINT "zu gross."
50 GOTO 20
```

Die Einrückungen werden vom BASIC-Interpreter nicht ausgewertet, sondern dienen nur dazu, optisch darzustellen, welches THEN und ELSE zu welcher IF-Abfrage gehört. Der BASIC-Interpreter benutzt leider einen etwas unglücklichen Algorithmus, um festzustellen, welches ELSE zu welchem IF gehört.

Trifft eine Bedingung nicht zu, so wird der erste gefundene ELSE-Pfad abgearbeitet, außer wenn im übersprungenen THEN-Pfad eine weitere Fallabfrage erfolgte. Dann nimmt der BASIC-Interpreter an, daß ein nachfolgendes ELSE sich auf dieses IF bezieht. Im folgenden Programm kann der zweite ELSE-Pfad deshalb nie abgearbeitet werden, auch wenn durch das Einrücken sinnfällig gemacht ist, daß, wenn der erste Test (a<=b) fehlschlägt, das zweite ELSE bearbeitet werden soll:

```
 5 REM Dieses Vergleichsprogramm funktioniert nicht im Schneider-BASIC
 6 REM
10 INPUT "zwei Zahlen bitte:",a,b
20 IF a<=b THEN IF a=b THEN PRINT "a und b sind gleich"
                       ELSE PRINT "a ist kleiner als b"
          ELSE PRINT "a ist groesser als b"
30 END
```

Andere BASIC-Interpreter oder auch PASCAL sind da besser:

```
program vergleich (input,output);
  var a,b:integer;
  begin
    writeln ('zwei Zahlen bitte:'); readln (a,b);
    if a<=b then if a=b then writeln ('a und b sind gleich')
                        else writeln ('a ist kleiner als b')
            else writeln ('a ist groesser als b')
  end.
```

Pascal macht es sich leicht, indem nach then und else nur ein einziger Befehl folgen darf, der bedingungsabhängig abgearbeitet werden soll. Werden mehr benötigt, dann können beliebig viele Einzeloperationen mit begin und end zu einem einzigen Befehl geklammert werden. Die innere Fallabfrage im obigen Beispiel "if a=b then ... else ..." zählt dabei in Pascal als ein Befehl und muß deshalb nicht mit begin und end geklammert werden.

Jede IF-Verzweigung kann nach einem einfachen Schema in bedingte Sprünge zerlegt werden:

```
 4 REM HI-LO-Spiel
 5 REM -----------
10 z=INT(RND*1000)
20 INPUT "rate:",e
30 IF e=z THEN PRINT "richtig.":GOTO 10
          ELSE IF e<z THEN PRINT "zu klein."
                      ELSE PRINT "zu gross."
50 GOTO 20

Zeile 30 ——>

30 CASE (e=z) : IFTRUEGOTO 38
32 CASE (e<z) : IFTRUEGOTO 36
34 PRINT "zu gross":GOTO 50
36 PRINT "zu klein":GOTO 50
38 PRINT "richtig.":GOTO 10 :GOTO 50
```

Da es in BASIC kein echtes bedingtes GOTO gibt, wurde es für dieses Beispiel extra erfunden. Mit CASE wird eine Aussage in falsch oder richtig ausgewertet und irgendein Flag entsprechend gesetzt. Der bedingte Sprung IF-TRUEGOTO erfolgt nur, wenn das Flag auf Wahr steht.

INDIREKTER SPRUNG – VERZWEIGUNG VIA TABELLE

Wenn mehrere Fälle zu unterscheiden sind, empfiehlt sich oft ein indirekter Sprung. Hierbei muß sich aus der Bedingung ein Zeiger berechnen lassen, mit dessen Hilfe aus einer Tabelle das Sprungziel ausgelesen werden kann.

In BASIC steht hierfür der Befehl "ON bedingung GOTO" zur Verfügung. Besonders in Menüs wird man oft davon Gebrauch machen:

```
10 PRINT " 1 = Laden  2 = Speichern  3 = Ueberprüfung  4 = Exit"
20 INPUT " bitte wählen Sie.",wahl
30 IF wahl<1 OR wahl>4 THEN GOTO 20
40 ON wahl GOTO 2000,3000,4000,5000
...
```

Hierbei wird in Zeile 30 zuerst getestet, ob die Eingabe im gültigen Bereich liegt. In BASIC wäre eine Überschreitung des Bereichs nicht schlimm. Ist der Wert zu groß, so macht BASIC mit dem nächsten Befehl weiter, was sehr nützlich sein kann, wenn man sehr umfangreiche Menüs zu bearbeiten hat:

```
 5 REM Verteiler in einem Textverarbeitungsprogramm
 6 REM ------------------------------------------
10 LET i$ = INKEY$: IF i$="" THEN GOTO 10
20 LET wahl=ASC(i$)
30 IF wahl >= 32 THEN GOTO 5000 : REM kein Controlcode ->
Zeichen drucken
40 '
50 Controlcode behandeln
60 '
70 ON wahl+1  GOTO  200, 300, 400, 500, 600, 700, 800, 900
80 ON wahl-7  GOTO 1000,1100,1200,1300,1400,1500,1600,1700
90 ON wahl-15 GOTO 1800,1900,2000,2100,2200,2300,2400,2500
95 ON wahl-23 GOTO 2600,2700,2800,2900,3000,3100,3200,3300
...
```

Bei einer Bereichsunterschreitung um eins macht BASIC auch mit dem nächsten Befehl weiter. Bei mehr bleibt der BASIC-Interpreter aber mit einer Fehlermeldung stehen:

```
ON 0 GOTO 200 [ENTER]
Ready

ON -1 GOTO 200 [ENTER]
Improper argument
Ready
```

Vor allem aber in Assembler- oder compilierten Programmen ist diese sogenannte "Plausibilitätskontrolle" unerläßlich. In Assembler muß man sich eben um fast alles selbst kümmern. Wenn man nicht kontrolliert, daß der "wahl"-Zeiger auch wirklich in die Tabelle möglicher Sprungadressen zeigt und nicht darüber oder darunter, landet das Programm an einer unbestimmten Stelle, sobald der Anwender ungültige Eingaben macht:

```
; Verzweigung via Tabelle in Maschinencode:
; -----------------------------------------

MENUE:  LD    HL,MENTXT      ; Menue auf dem Bildschirm ausgeben.
        CALL MESSEG
```

```
MEN1:   CALL INKEY          ; Warte, dass Anwender eine Taste drückt.

        SUB  "0"            ; ASCII-Code für "0" bis "9"
                            ; in ein Byte 0 .. 9 wandeln und
        JR   C,MEN1         ; falls Code zu klein
;                           ; (kleiner als Code für '0')
        CP   10             ; Teste, ob Code zu gross (groesser als '9')
        JR   NC,MEN1

        ADD  A,A            ; Zeiger in die Adressen-Tabelle berechnen
        LD   HL,BASIS       ; (Basis + 2*Index = Zeiger auf Sprungadresse)
        LD   E,A
        LD   D,0
        ADD  HL,DE

        LD   E,(HL)         ; Adresse aus der Tabelle nach Register DE
;                           ; holen
        INC  HL
        LD   D,(HL)

        EX   DE,HL          ; Routine anspringen
        JP   (HL)

MENTXT: DEFM ".... 10 MENUE-OPTIONEN MIT NUMMERN VON 0 BIS 9 ... "
        DEFB 0

BASIS:  DEFW ADR0           ; Tabelle mit 10 Adressen (je 2 Bytes) für die
        DEFW ADR1           ; Behandlungsroutinen der 10 Menue-Optionen
        ....
        DEFW ADR9
...
```

ITERATIONEN – SCHLEIFEN

Eins der einfachsten Strukturierungsmittel ist die Schleife. Der Befehlssatz des Z80 (und praktisch jedes anderen Mikroprozessors auch) enthält keine expliziten Schleifenbefehle. Wer in Assembler programmiert, muß sich seine Schleifen immer mit Hilfe von bedingten Sprüngen selbst programmieren.

Um Schleifen optisch besser erkennbar zu machen, werden die Befehle innerhalb der Schleife oft eingerückt. Aber praktisch keine Programmiersprache wertet dieses Einrücken auch aus. Es dient ausschließlich der besseren Überschaubarkeit.

Schleifen dienen dazu, den in ihnen enthaltenen Programmteil mehrfach auszuführen, und müssen normalerweise ein Endkriterium haben. Fehlt dies,

wird die Schleife unendlich oft durchlaufen. Kommen in einem Programm mehrere Schleifen vor, so dürfen sie nicht "verschränkt" werden:

Erlaubt:

aufeinanderfolgende Schleifen:

```
prog: befehl
      befehl
      start 1 <-+
        befehl  |
        befehl  |
        befehl  |
      ende 1  --+
      befehl
      start 2 <-+
        befehl  |
      ende 2  --+
      ...
```

Erlaubt:

verschachtelte Schleifen:

```
prog: befehl
      start 1 <-+
        befehl  |
        befehl  |
        start 2 <-+
          befehl  |
          befehl  |
        ende 2  --+
      ende 1  --+
      befehl    |
      befehl    |
      ...       |
```

Unsinnig:

verschränkte Schleifen:

```
prog: befehl
      befehl
      start 1 <-+
        befehl  |
        befehl  |
        start 2 <-+
          befehl  |
          befehl  |
      ende 1  --+
      befehl    |
        ende 2  --+
        ???
```

Das BASIC der Schneider CPCs kennt nur zwei unterschiedliche Schleifenstrukturen:

```
100 FOR A=1 TO 10 STEP 1
110 PRINT A
120   NEXT A
130 END
```

oder:

```
200 A=1
210 WHILE A>0
220   PRINT A
230   A=A/2
240 WEND
250 END
```

Die FOR/NEXT-Schleife bietet dabei den zusätzlichen Service, eine sogenannte *Laufvariable* nach jedem Schleifendurchgang automatisch weiterzustellen. Im Schleifenkopf muß dabei der Laufbereich (von...bis) und die Schrittweite angegeben werden. Das Schleifenende wird durch den Befehl NEXT festgelegt.

FOR/NEXT-Schleifen weisen leider von BASIC zu BASIC zum Teil erhebliche Unterschiede auf. Deshalb sind sie bei "strukturierten" Programmierern auch nicht sehr beliebt.

Laufindex

```
10 FOR i=1 TO 10:PRINT i:NEXT i
20 PRINT i
```

Welchen Wert hat im obigen Programm die Variable I nach dem letzten Schleifendurchlauf? Wenn Sie das Programm abtippen, können Sie sich davon überzeugen, daß I zum Schluß den Wert 11 enthält. Bei manchen anderen Interpretern ist aber auch der Wert 10 möglich. Das hängt davon ab, wie der Interpreter das Schleifenende testet: zuerst die Laufvariable weiterstellen und dann auf Überschreitung der Grenze testen (wie Locomotive BASIC im CPC) oder erst auf Erreichen der Grenze testen und nur gegebenenfalls den Laufindex weiterstellen?

Struktur

Ein weiterer Unterschied ergibt sich in der Art, wie der Interpreter den Zusammenhalt zwischen dem Schleifenkopf (FOR-Statement), dem Schleifenende (NEXT) und der Laufvariablen wahrt.

Locomotive BASIC trägt in seinem privaten Return- und Schleifenstapel jede gestartete Schleife ein. Der Eintrag enthält unter anderem einen Zeiger auf das FOR-Statement und einen Zeiger auf NEXT. Es ist deshalb nicht möglich, einer Schleife zwei verschiedene Schleifen-Endpunkte zuzuordnen, was beispielsweise nach einer Fallabfrage interessant sein könnte.

Das folgende Programm läuft auf dem Schneider CPC nicht, beispielsweise aber auf dem Sinclair SPECTRUM:

```
 80 INPUT "Startwert";S
 90 INPUT "  Endwert";E
100 FOR A=S TO E
110   IF INT(A/77) <> A/77 THEN NEXT A:END
120   PRINT A;"ist durch 77 teilbar"
130 NEXT A
140 END
```

Beim Spectrum werden die Schleifenparameter nämlich an die Laufvariable gebunden. Hier ist die NEXT-Adresse nicht angegeben, nur die Adresse des FOR-Statements. Dadurch verzweigt dessen BASIC bei jedem NEXT A zur entsprechenden FOR-Deklaration, wenn die Schleifenbedingung erfüllt ist.

Das folgende Programm kann, je nach BASIC-Interpreter, recht unterschiedliche Wirkungen entfalten. Die Wirkung wird aber nur in den seltensten Fällen die sein, die man von ihm erwartet:

```
10 FOR A=10 TO 100 STEP 10
20   FOR A=A TO A+9 STEP 1
30     PRINT A
40   NEXT A
50 NEXT A
```

Abweisende Schleifen

Alle Schleifen lassen sich in zwei unterschiedliche Kategorien einordnen: die abweisenden Schleifen, bei denen der Schleifeninhalt kein einziges Mal abgearbeitet wird, wenn das Abbruchkriterium bereits vor dem ersten Durchlauf erreicht ist, und die nicht abweisenden Schleifen, deren Inhalt mindestens einmal durchlaufen wird, bevor eine Endabfrage über weitere Wiederholungen entscheidet.

Im BASIC des Schneider CPC ist die zweite Form realisiert, was folgendes Beispiel zeigt:

```
100 FOR A=100 TO 99 STEP +1
110   PRINT A
120 NEXT A

RUN [ENTER]
Break in 130
Ready
```

In diesem Fall wird die Schleife kein einziges Mal durchlaufen und auch der Schleifenindex nicht vor der Endabfrage erhöht. A enthält nach "Durchlauf" der Schleife den Wert 100.

Für die WHILE/WEND-Schleife gibt es da schon verläßlichere Kriterien. Auch sie ist eine abweisende Schleife. Der Schleifeninhalt wird so lange durchlaufen, wie die im Schleifenkopf angegebene Bedingung erfüllt ist (while – deutsch: während).

Während unser BASIC also nur abweisende Schleifen kennt, muß man hierfür in Assembler zusätzlichen Aufwand treiben:

nicht abweisend:

```
START:  LD   B,0
LOOP:   PUSH BC
```

abweisend:

```
START: LD   B,0
       INC  B
```

nicht abweisend:

```
START: LD      B,0
       LD      A, "*"
LOOP:  CALL    #BB5A
       DEC     B
       JR NZ, LOOP
       RET

ANM.: #BB5A = TXT OUTPUT
              Druckt Zeichen in A
```

abweisend:

```
START: LD      B,0
       LD      A, "*"
       INC     B
       JR      ENDE
LOOP:  CALL    #BB5A
ENDE:  DEC     B
       JR      NZ, LOOP
       RET
```

Hier ist der Unterschied besonders kraß. Das Ergebnis beider Schleifen ist dasselbe, solange der Startwert für B nicht mit null festgesetzt wird. Dann erhält man nämlich in der nicht abweisenden Version 256 Sternchen, während die abweisende Version kein einziges druckt.

UNTERPROGRAMME

Das wichtigste aller Strukturierungsmittel überhaupt stellen die Unterprogramme dar. Nur mit ihnen ist es möglich, einzelne Befehlsfolgen, die in einem Programm mehrmals vorkommen, nur einmal im Speicher des Computers niederzuschreiben und trotzdem beliebig oft auszuführen. Im Gegensatz zu Schleifen können diese Befehlssequenzen von jeder beliebigen Stelle des Programms aus aufgerufen werden.

Programmtechnisch gesehen stellt ein Unterprogramm-Aufruf einen Sprung dar. Beim normalen Sprung wird der Befehlszeiger sofort mit der Sprungadresse geladen. Die Stelle, ab der der Sprung erfolgte, wird vergessen.

Beim Unterprogramm-Aufruf wird diese Adresse aber gerettet. Bevor der Befehlszeiger mit der neuen Adresse geladen wird, wird sein alter Wert auf einem Stapel abgelegt. Dann erst wird der Befehlszeiger verändert. Danach wird das Programm wie bei einem normalen Sprung ab der neuen Adresse abgearbeitet.

Ist die Behandlung des Unterprogramms abgeschlossen, so muß ein spezieller Rücksprungbefehl abgearbeitet werden, der den alten Wert wieder vom Stack herunternimmt und in den Befehlszeiger zurücklädt. Danach wird das alte Programm genau ab der Stelle fortgesetzt, an der der Unterprogramm-Aufruf erfolgte. Da die Rücksprungadressen auf einem Stapel abgelegt werden, kann man Unterprogramme auch schachteln: Wird in einem Unterprogramm ein

weiteres Unterprogramm aufgerufen, so wird diese Rücksprungadresse ebenfalls auf den Stapel gelegt. Jeder Rückkehrbefehl nimmt immer nur die oberste Adresse vom Stapel. Ob darunter noch eine weitere Rückkehradresse liegt, ist so lange uninteressant, bis ein weiterer RETURN-Befehl abgearbeitet wird.

In BASIC wird für den Aufruf der Befehl GOSUB und für den Rücksprung RETURN benutzt. Der Stapel ist dabei (außer mit PEEK und POKE) nicht zugänglich.

In Assembler werden die Mnemonics CALL und RET gebraucht. Der Maschinenstapel der CPU wird dabei für die Rücksprungadressen und zum Zwischenspeichern von Registern mit PUSH und POP benutzt. Die Return-Adressen sind deshalb sehr leicht zugänglich, weil das Unterprogramm jeden Stack-Eintrag in ein Doppelregister "poppen" kann. Für Manipulationen sind hier Tür und Tor geöffnet.

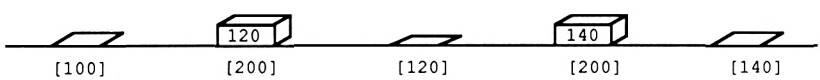
Die folgenden Programme und Grafiken zeigen das Verhalten von Unterprogrammen und die Auswirkungen auf den Stapel:

```
100 A=10:B=5
110 GOSUB 200
120 A=8:B=9
130 GOSUB 200
140 STOP
200 PRINT A;"*";B;"=";A*B
210 RETURN
```

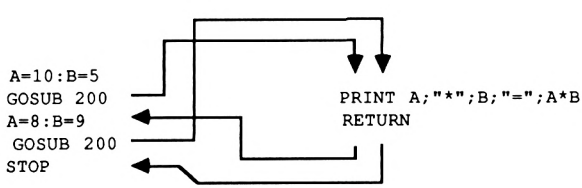
Reihenfolge der Bearbeitung:

```
[100] [110]  [200] [210]
[120] [130]  [200] [210]
[140]
```

Einträge auf dem Stack:



Grafische Darstellung:



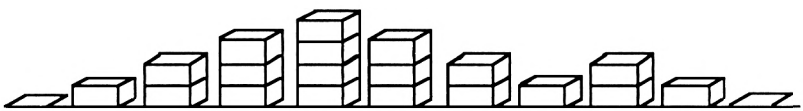
```
 90        ...
100 PROG: PUSH  BC
110       PUSH  DE
120       CALL  UP1
130       POP   DE
140       CALL  UP2
150       POP   BC
160       ...

200       CALL  UP2
210       RET
300       RET
```

Reihenfolge der Bearbeitung:

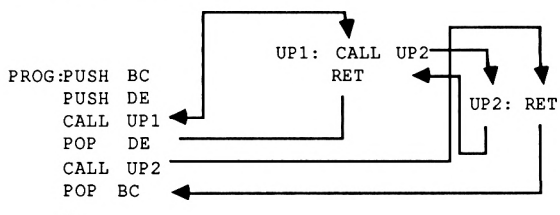
```
[100] [110] [120] [200] [300] [210]
[130] [140] [300] [150]
```

Einträge auf dem Stack (jeweils nach dem Befehl):



```
[90] [100] [110] [120] [200] [300] [210] [130] [140] [300] [150]
```

Grafische Darstellung:



Unterprogramme spielen bei der Modularisierung von Problemen eine ganz große Rolle. Die einzelnen Probleme werden nämlich nicht einfach hintereinander gehängt, sondern als Unterprogramme gelöst, die von übergeordneten Modulen aufgerufen werden können.

Das hat den Vorteil, daß einzelne Module mehrfach verwendet werden können. Als Unterprogramme kann man von jeder beliebigen Stelle auf sie zugreifen.

Vor allem die primitivsten Routinen werden am häufigsten aufgerufen. So muß man sich beispielsweise klarmachen, daß die Ausgabe von Buchstaben auf dem Bildschirm auch von einem Unterprogramm erledigt wird. Ein einfaches CALL auf die entsprechende Routine der Text-VDU: Alles weitere erledigt dieses Modul, und hinterher steht das Zeichen auf dem Bildschirm.

Und das ist wieder das Kennzeichen strukturierter Programmierung. Interessant ist nur die Schnittstellen-Beschreibung:

Eingaben – Funktion – Ausgaben

VARIABLEN

Neben dem Programmablauf ist die Speicherung von Variablen ein wichtiger Punkt. Es handelt sich dabei um die oberste Ebene der "Datenspeicherung".

Eine der Hauptaufgaben von Hochsprachen ist es, den Programmierer vom Problem der Variablen-Speicherung zu entlasten. BASIC-Programmierer brauchen sich überhaupt nicht damit zu befassen. Für sie kann man Variablen einrichten, indem man einem Namen einfach einen Wert zuweist:

```
100 LET A=123.456
```

Dafür stellt BASIC aber auch keinen besonders hohen Komfort zur Verfügung. Es kennt nämlich nur globale Variablen.

Globale Variablen

Auf globale Variablen kann man, im Gegensatz zu lokalen Variablen, vom gesamten Programm aus zugreifen. Hat das Hauptprogramm beispielsweise in der Variablen A einen wichtigen Wert gespeichert, so ist es programmtechnisch unmöglich zu verhindern, daß irgendein Unterprogramm diese Variable A ebenfalls benutzt und verändert. Um das zu verhindern, könnte man im Unterprogramm nur Variablen mit anderen Namen benutzen.

Es ist aber ein wesentlicher Gesichtspunkt bei der Modularisierung von Problemen, daß die Variablen eines Programms erhalten bleiben, wenn es Unterprogramme aufruft. In BASIC muß der Programmierer des Hauptprogramms immer in gewissem Umfang über das Innenleben der benutzten Unterprogramme Bescheid wissen. Die dort vorkommenden Variablen dürfen sich nicht mit den von ihm benutzten überschneiden!

Lokale Variablen

Dieses Problem der "Datenkapselung" ist nur mit lokalen Variablen zu lösen. Die aufgerufenen Programm-Module müssen sich ihre eigenen Variablen einrichten, vollkommen unabhängig von bereits bestehenden Variablen, die mög-

licherweise den gleichen Namen haben. Es darf den Modulen gar nicht möglich sein, Variablen der rufenden Programme zu verändern. Wenn überhaupt, dürfen dafür nur genau definierte Befehle vorgesehen sein.

Lokale Variablen werden deshalb immer auf einem Stapel eingerichtet. Oftmals ist das sogar direkt der Prozessorstapel, auf dem auch die Rücksprungadresse des Modulaufrufs abgelegt wird.

Auch in BASIC lassen sich lokale Variablen simulieren, wenn man sich einen Stapel bastelt. Dazu benötigt man ein Zahlenfeld und einen Zeiger:

```
100 DIM stk(100):sp=0        ' Stapel einrichten und Stapelzeiger
                               initialisieren
110 FOR i=0 TO 100 STEP 10   ' Variable 'i' wird im Hauptprogramm benutzt
120 eingabe=i                ' Eingabe für Modul in 'eingabe'
130 GOSUB 200                ' Modul-Aufruf
140 PRINT ausgabe            ' Ausgabe des Moduls in 'ausgabe' ausdrucken
150 NEXT
160 END

200 stk(sp)=i:sp=sp+1          ' Variable 'i' retten (PUSH i)
210 stk(sp)=j:sp=sp+1          ' Variable 'j' retten (PUSH j)
220 j=0                        ' lokales 'j' kann geändert werden
230 FOR i=eingabe TO eingabe+9 ' lokales 'i' wird geändert. 'eingabe'
                                 benutzt
240 j=j+PEEK(i)
250 NEXT
260 ausgabe=j                  ' Ausgabe in Variable 'ausgabe'
270 sp=sp-1:j=stk(sp)          ' Variable 'j' restaurieren (POP j)
280 sp=sp-1:i=stk(sp)          ' Variable 'i' restaurieren (POP i)
290 RETURN
```

Die lokalen Variablen wurden simuliert, indem das Modul alle Variablen, die es benutzte, zuerst auf den Stack gerettet hat. Danach konnten die (tatsächlich immer noch globalen) Variablen "lokal" verändert werden. Nach Abschluß der Routine wurden die ursprünglichen Werte wieder in die veränderten Variablen zurückgeschrieben, so daß dem aufrufenden Programm keine Daten verlorengehen konnten.

In Maschinensprache steht aber bereits ein Stapel zur Verfügung, auf dem auch Register abgelegt werden können. Die obige Methode ist in Assembler sehr leicht zu verwirklichen:

```
TEST:   LD   HL,TEXT1
        CALL MELDNG
        RET
;
MELDNG: PUSH AF
        CALL MELD1
```

```
        POP   AF
        RET
;
MELD1:  LD    A,(HL)
        INC   HL
        AND   A
        RET   Z
        CALL  #BB5A  ; TXT OUTPUT
        JR    MELD1
;
TEXT1:  DEFM "Hallo Schneider-User!"
        DEFB 0
```

Das Unterprogramm MELD1 enthält eine Schleife, die HL als Zeiger auf einen auszudruckenden Text benutzt. Um die Zeichen auszudrucken, wird der Vektor TXT OUTPUT aufgerufen. Wird nicht MELD1, sondern MELDNG aufgerufen, so wird für die Dauer des Unterprogramms das Doppelregister AF (Akku und Flags) gerettet.

TXT OUTPUT rettet auf entsprechende Weise alle Register (AF, BC, DE und HL), bevor es selbst wieder eine andere Routine aufruft, die das Zeichen ausdruckt. Deshalb muß auch der Zeiger HL nicht vom rufenden (Unter-) Programm MELD1 zwischengespeichert werden.

Für das Modul MELDNG gilt also folgende Schnittstellen-Beschreibung:

Funktion: Gibt einen Text im aktuellen Bildschirmfenster aus. Control-Zeichen werden befolgt (weil dies auch TXT OUTPUT tut).

Eingaben: HL zeigt auf das erste Zeichen des Textes. Der Text muß mit einem Nullbyte abgeschlossen sein.

Ausgaben: HL zeigt hinter den Text (auf das Nullbyte).

Unverändert: AF, BC, DE, IX und IY.

ÜBERGABE VON ARGUMENTEN UND ERGEBNISSEN

In BASIC ist es nicht vorgesehen, Argumente an Unterprogramme zu übergeben. Man kann sich aber leicht behelfen, indem man dafür zwischen rufendem und gerufenem Programm eine Variable vereinbart. Diese Methode, mit den Variablen eingabe und ausgabe wurde in dem letzten BASIC-Beispiel mit den "lokalen" Variablen verwendet.

Eine Ausnahme bilden aber die vom Programmierer definierbaren Funktionen:

```
10 DEF FNmittel(a,b) = (a+b)/2
20 '
30 FOR i=1 TO 90
40   PRINT FNmittel(SIN(i),COS(i))
50 NEXT i
60 END
```

An eine Funktion kann man Argumente übergeben (in diesem Fall SIN(i) und COS(i)), die in die Formel der Funktion anstelle der "Platzhalter" a und b eingesetzt werden. Die Variablen a und b sind dabei lokale Variablen.

Andere Programmiersprachen haben genau festgelegte Strukturen, mit denen an ein Unterprogramm Werte übergeben und auch wieder übernommen werden können:

LOGO

```
to mittelwert :a :b
op (:a + :b)/2
end
```

Definition der Prozedur und von a und b als lokale Übernahmevariablen für zwei Parameter.
Ausgabe von (a+b)/2 als Ergebnis der Prozedur.

Aufruf:

```
mittelwert 100 200 [ENTER]
150
```

Pascal

```
function mittelwert (a,b:real):real;

begin
mittelwert := (a+b)/2

end;
```

Definition der Funktion und von a und b als lokale Übernahmevariablen für zwei Parameter.
Ausgabe von (a+b)/2 als Ergebnis der Funktion.

Aufruf:

```
writeln (mittelwert(100,200))
```

FORTH

```
: mittelwert
+
```

Beginn der Definition des Wortes mittelwert
Addiere die beiden obersten Stack-Einträge.

```
2          Push die Zahl 2 auf den Stapel.
/          Dividiere den vorletzten durch den letzten Stack-Eintrag.
;          Ende der Definition
```

Aufruf:

```
100 200 mittelwert . [ENTER]
150
```

In LOGO und Pascal bedarf es einer speziellen Syntax, um eine Funktion als Parameter-nehmend und Ergebnis-liefernd zu deklarieren. Dafür kontrolliert der Interpreter (LOGO) bzw. Compiler (Pascal) bei jedem Aufruf der Funktion, ob diese Parameterschnittstelle auch eingehalten wird.

In FORTH, das grundsätzlich alle Operationen auf seinem Variablen-Stack durchführt, nehmen alle Worte (FORTH-Prozeduren) die Argumente von diesem Stapel und legen das Ergebnis hinterher wieder darauf ab. Vor einem Aufruf eines Wortes müssen die Parameter also auf dem Stapel abgelegt werden. Eine Kontrolle, ob das auch wirklich bei jedem Aufruf geschieht, gibt es nicht. Allenfalls werden vom FORTH-Compiler Tests auf Unterschreiten des Stapelbodens in die compilierten Worte eingefügt.

In Assembler ist es zumindest bei der Z80-CPU üblich, Argumente in Registern zu übergeben. Bei komplizierteren Datentypen, wie beispielsweise Strings oder Fließkomma-Zahlen, werden die Register nur als Zeiger auf die Argumente verwendet. Im Prinzip ist man aber ungebunden, was die Methode der Parameterübergabe betrifft. Man sollte sich nur um ein möglichst einheitliches Verfahren bei allen Unterprogramm-Modulen bemühen, damit man nicht hinterher in einem Programm nach einer fehlerhaften Parameterübergabe suchen muß.

Die Arithmetikroutinen des CPC-Betriebssystems können hierbei als Beispiel dienen. Fast alle befolgen folgende Schnittstellenbeschreibung:

Eingabe: HL = erstes Argument, bei Integer direkt, bei Real ein Zeiger.
DE = zweites Argument, falls nötig (bei Operationen).

Ausgabe: HL = Ergebnis.

Als Beispiel wird im folgenden Programm das Mittelwertproblem auch noch in Assembler gelöst. Eingaben sind (System-konform) das HL- und das DE-Register, in denen Zeiger auf zwei Fließkomma-Zahlen erwartet werden.

```
ADD:    EQU  #BD58          ; CPC 664: #BD79  ; CPC 6128: #BD7C
DIV:    EQU  #BD64          ; CPC 664: #BD85  ; CPC 6128: #BD88
;
MITTEL: CALL ADD            ; Addiere FLO(HL) := FLO(HL) +
```

```
;                               ; FLO(DE)
        RET   NC                ; überlauf
        LD    DE,ZWEI           ; FLO(DE) := 2
        CALL  DIV               ; Dividiere
;                               ; FLO(HL) := FLO(HL) / FLO(DE)
        RET
;
ZWEI:   DEFB  0                 ; 2 in der internen Fliesskomma-
;                               ; Darstellung
        DEFB  0
        DEFB  0
        DEFB  0
        DEFB  130
```

Eine Fließkomma-Zahl zerlegt man leicht mit folgendem Trick in die fünf Bytes für ihre interne Codierung:

```
10 LET a=2
20 FOR i=@a TO @a+4
30 PRINT PEEK(i);
40 NEXT
```

In Zeile 10 wird eine Variable mit dem gewünschten Wert definiert, worauf der BASIC-Interpreter diese Variable im Speicher einrichtet und die Zahl in der internen Codierung abspeichert. Hier können die fünf Bytes dann einfach mit Hilfe des Variablenpointers @ aus dem Speicher ausgelesen werden.

REKURSION – SELBSTAUFRUF EINER PROZEDUR

Ein bei allen Mathematikern beliebtes Verfahren stellt die Rekursion dar. Dabei löst eine Funktion ein Problem dadurch, daß sie das Problem verkleinert und sich wieder selbst aufruft, um von sich selbst das verkleinerte Problem lösen zu lassen.

Wichtig ist, daß bei jedem rekursiven Aufruf auch wirklich das Problem verkleinert wird, und daß es zu einem bestimmten Zeitpunkt direkt lösbar wird. Sonst gehen die Aufrufe so lange weiter, bis der Return-Stack voll ist (und, wenn hier keine Überlauf-Kontrolle stattfindet, so lange, bis sich das Programm endgültig aufgehängt hat).

Das Abbruchkriterium, mit dem entschieden wird, wann das Problem direkt gelöst wird, ist also der eine entscheidende Trick. Der andere ist, daß man sich mit jedem Selbstaufruf auch wirklich auf das Abbruchkriterium hinbewegt.

An die Programmiersprache werden dabei zwei Anforderungen gestellt: Zum einen muß der Selbstaufruf einer Prozedur überhaupt möglich sein, und zum anderen müssen lokale Variablen definierbar sein.

Letzteres ist der Grund dafür, daß die Rekursion in BASIC so selten angewendet wird. BASIC kennt ja nur globale Variablen. Trotzdem sind mit einigem Geschick auch hier Rekursionen möglich. Zur Not muß man sich einen privaten Stapel in einem Array konstruieren, auf den man die "lokalen" Variablen rettet.

Füllalgorithmus

Bestechend ist der folgende BASIC-Einzeiler, der einen 100% funktionsfähigen Füllalgorithmus für beliebig umrandete Flächen darstellt (hier für Bildschirmmodus 1, bei dem der Pixel-Abstand auf dem Monitor in X- und Y-Richtung je zwei Längeneinheiten beträgt):

```
10 IF TEST(x,y)>0 THEN RETURN
                  ELSE PLOT x,y: x=x-2:         GOSUB 10:
                                 x=x+4:         GOSUB 10:
                                 x=x-2: y=y-2:  GOSUB 10:
                                        y=y+4:  GOSUB 10:
                                        y=y-2:  RETURN
```

Diese Routine kommt ohne lokale Variablen aus, weil die Argumente des Aufrufs (die X- und Y-Koordinaten) nicht verändert werden: x–2+4–2=x und y–2+4–2=y.

Abbruchkriterium ist der Test, ob der Punkt (x,y) gesetzt ist oder nicht. Ist der Punkt gesetzt, also TEST(x,y)>0, so ist die Umrandung erreicht. Der Punkt wird nicht noch einmal neu gesetzt, die Routine kehrt sofort zurück.

Ist das Pixel aber noch nicht gesetzt, so wird auf diese Stelle geplottet, und dann werden nacheinander die vier benachbarten Punkte ebenfalls getestet, wozu vier rekursive Aufrufe nötig sind.

Dem praktischen Einsatz dieser Füllfunktion steht leider im Weg, daß sie sich, je nach Größe der auszumalenden Fläche, mehrere tausendmal selbst aufrufen kann. Das überfordert den Return-Stack des BASIC-Interpreters.

Fakultät

Ein weiteres beliebtes Beispiel ist die rekursive Berechnung der Fakultät. Die Fakultät n! einer ganzen, positiven Zahl n ist das Produkt aus allen ganzen Zahlen ab 1 bis zu dieser Zahl n selbst:

n! = 1*2*3*4* ... (n–2)*(n–1)*n

Die Fakultät von 0 ist per Definition 1. In BASIC wird man das Problem sehr wahrscheinlich mit einer Schleife angehen:

Fakultät mittels Iteration

```
10 INPUT "eine Zahl:",n
20 LET fak=1
30 FOR n=1 TO n:fak=fak*n:NEXT n
40 PRINT fak
50 GOTO 10
```

und mittels Rekursion

```
10 INPUT "eine Zahl:",n
20 GOSUB 40:PRINT fak:GOTO 10
30 '
40 IF n<2 THEN fak=1:RETURN
50 n=n-1:GOSUB 40:n=n+1:fak=fak*n
   :RETURN
```

Die Rekursion basiert auf dem Gedanken, daß die Fakultät einer Zahl n sich auf die Fakultät der nächstkleineren Zahl n–1 zurückführen läßt:

n! = (n–1)! * n z.B.: 4! = 1*2*3*4 = (1*2*3) * 4 = 3! * 4

Tatsächlich läßt sich jede Rekursion auf eine Iteration (Schleife) zurückführen. In aller Regel gilt dabei: Die Iteration ist schneller und verbraucht weniger Speicherplatz. Der Vorteil der Rekursion ist aber, daß man hier mechanisch beweisen kann, daß die Rekursion terminiert (zu einem Ende kommt) und daß manches Problem hier schlicht viel einfacher zu lösen ist.

Das Abbruchkriterium ist der Test in Zeile 40, ob n kleiner als 2 wird. Dann ist die Fakultät 1 weil:

0! = 1 und 1! = 1

Ist das Abbruchkriterium noch nicht erreicht, so wird in Zeile 50 die Fakultät rekursiv bestimmt: n wird um 1 erniedrigt und davon die Fakultät errechnet. Diese mit n multipliziert ergibt, wie oben erklärt, n!. Dabei darf n selbst nicht verändert werden, weil n in BASIC ja nicht lokal definiert werden konnte.

Das folgende Pascal-Programm berechnet die Fakultät mit lokalen Variablen:

```
1 function fak (n:integer):integer;
2 begin
3   if n<2 then fak := 1
4          else fak := fak(n-1)*n
5 end;
```

In der ersten Zeile wird u.a. die lokale Übernahmevariable n definiert. Zeile 3 ist wieder der Test auf das Abbruchkriterium. Ist es noch nicht erreicht, so wird in Zeile 4 die Fakultät rekursiv ermittelt. Dabei wird das Argument n–1

zur neuen lokalen Übernahmevariablen n der aufgerufenen Funktion fak. Wie man sieht, konnte wegen der nur lokalen Gültigkeit der Übernahmevariablen n die zugrundeliegende Formel viel direkter übernommen werden:

```
fak(n) := fak(n-1)*n
```

Ebenfalls einsichtiger wird der Füll-Einzeiler, wenn man ihn in Pascal schreibt:

```
1 procedure fuell (x,y:integer);
2 begin
3   if test(x,y)=0 then begin
4                       plot(x,y);
5                       fuell(x-2,y); fuell(x,y-2);
6                       fuell(x+2,y); fuell(x,y+2)
7                       end
8 end;
```

INTERRUPTS – UNTERBRECHUNGEN

Mit dem programmierbaren Interrupt-Mechanismus, der beim Schneider CPC in geradezu einzigartiger Weise bis zur Hochsprachenebene durchgeführt ist, kann man sehr elegant mehrere Aufgaben quasi gleichzeitig vom Computer erledigen lassen. Funktionen, die normalerweise mit längeren Wartezeiten verbunden sind, werden nur noch bei Bedarf aufgerufen. In der Zwischenzeit kann das Hauptprogramm ablaufen, ohne daß Rechenzeit in Warteschleifen verbraucht wird.

Sobald ein Interrupt-Signal auftritt, wird die zugehörige Routine wie ein normales Unterprogramm aufgerufen. Das heißt, die aktuelle Position des Befehlszeigers wird auf den Return-Stapel gerettet und der Befehlszeiger mit der Adresse der Interrupt-Routine geladen. Dadurch ist der Sprung zu dieser Routine realisiert, und diese kann nachher ganz normal mit RETURN abschließen.

Da Interrupts jederzeit unvorhersehbar im laufenden Hauptprogramm auftreten können, dürfen sie keine Register (Hardware-Interrupt in Maschinencode-Programmen) bzw. keine Variablen des Hauptprogramms (z.B. in BASIC) verändern.

Außerdem wird bei jedem Interrupt ein Flag gesetzt, das weitere Interrupts verhindert (EI und DI in Assembler). Der Software-Interrupt-Mechanismus des Kernel bietet darüber hinaus noch die Möglichkeit, einzelne Interrupts mit Prioritäten zu versehen, so daß "dringendere" Interrupts einem "unwichtigeren" Interrupt vorgezogen werden.

Sollen mit Hilfe von Interrupts mehrere Aufgaben (Tasks) quasi gleichzeitig abgearbeitet werden, so muß man sein Programm wie folgt gliedern:

Zunächst gibt es ein Hauptprogramm, das scheinbar ohne Unterbrechung abgearbeitet wird. Es unterscheidet sich in keiner Weise von einem Interrupt-freien Programm.

Aufgaben, die mit längeren Wartezeiten verbunden sind, können parallel dazu als Interrupt-Routinen ablaufen. Dabei muß man die Aufgabe so zerlegen, daß das Unterprogramm mit Beginn der Wartezeit zum Hauptprogramm zurückkehrt. Das Kriterium, das das Ende der Wartezeit anzeigt, muß benutzt werden, um einen Interrupt zu programmieren, der das Unterprogramm erneut aufruft.

Hintergrunduhr

Ein Beispiel für einen sinnvollen Einsatz des EVERY-Interrupt-Befehls in BASIC ist eine mitlaufende Uhr. Einmal pro Sekunde soll die Uhrzeit ausgegeben werden. Danach tritt eine Wartezeit auf, die so lange dauert, bis die Zeit erneut ausgegeben werden muß, also bis zum Eintritt der nächsten Sekunde:

```
10 INPUT "Uhrzeit [hh,mm,ss] = ",std,minu,sek
20 '
30 EVERY 50,1 GOSUB 1000 : REM starten der Uhr
40 '
50 GOTO 50 : REM Hauptprogramm

1000 sek=sek+1
1010 IF sek=60 THEN sek=0:minu=minu+1
1020 IF minu=60 THEN minu=0:std=std+1
1030 IF std=24 THEN std=0
1050 LOCATE#7,1,1 : PRINT#7,USING"##&###&##";std;":";
minu;":";sek;
1070 RETURN
```

In Zeile 30 wird ein Interrupt programmiert, der die Priorität 1 hat, alle 50/50 Sekunden auftreten soll und zum Aufruf des Uhrenprogramms ab Zeile 1000 führt. Hier werden dann die Sekunden und eventuell auch Minuten und Stunden weitergestellt.

Damit durch den Ausdruck der Zeit nicht die Textausgabe des Hauptprogramms gestört wird (und auch keine weiteren, möglicherweise installierten Interrupts die Textausgabe der Uhr stören), benutzt die Uhr ein eigenes Textfenster, das von keinem weiteren Programm benutzt werden darf.

Blinkender Cursor

Interessant ist auch ein blinkender Cursor z.B. in einer Textverarbeitung. Dabei wird der Cursor eingeschaltet, eine Weile gewartet, der Cursor ausgeschaltet und wieder gewartet, bevor der Zyklus von neuem beginnt:

```
 90 REM Hauptprogramm: Textverarbeitung 'einfachst'
 91 REM ------------------------------------------

100 GOSUB 500
110 i$=INKEY$:IF i$="" THEN GOTO 110
120 PRINT i$;
130 GOTO 100

490 REM Cursor-Blinken:

500 CALL &BB81          : REM Cursor On
510 AFTER 30 GOSUB 550
520 RETURN

550 CALL &BB84          : REM Cursor Off
560 AFTER 15 GOSUB 500
570 RETURN
```

Die beiden Routinen ab Adresse 500 und 550 rufen sich in stetem Wechsel gegenseitig auf und schalten dabei den Cursor-Fleck ein und wieder aus. Das Hauptprogramm (Zeilen 100 bis 130) merkt davon nichts. Dabei ist es übrigens eine gute Idee, nach jeder Tasteneingabe den Cursor zwangsweise einzuschalten, damit er sichtbar wird, wenn sich die Cursor-Position verschiebt. Das wird in diesem Beispiel dadurch erreicht, daß in Zeile 130 der Sprung nicht nur bis 110 zurückgeht, sondern bis Zeile 100, wo der Aufruf der Cursor-Einschaltroutine steht.

Während eines INPUT oder LINE INPUT werden die Interrupts leider nicht befolgt, da ihre Ausführung vom Pollen des BASIC-Interpreters abhängig ist. Zur genaueren Erklärung des Interrupt-Mechanismus im Schneider CPC folgen bei der Betrachtung der Firmware noch zwei eigenständige Kapitel.

Befehlselemente

Um die Arbeitsweise eines Compilers oder Interpreters zu verstehen, ist es besonders wichtig, sich mit den einzelnen Elementen auseinanderzusetzen, aus denen sich ein kompletter Befehl zusammensetzt. Wie kann der BASIC-Interpreter so komplizierte Anweisungen wie die folgende verstehen:

```
PLOT 100*(SIN(PI*r/180)+o),100-50*COS(delta),3
```

Was unterscheidet beispielsweise das Wort PLOT von SIN oder 180? Gemein ist den drei Zeichenfolgen, daß sie jeweils ein nicht mehr weiter teilbares Element der Sprache darstellen. Um die Unterschiede herauszuarbeiten, kann man sich einmal ihre "Schnittstellen"-Beschreibung ansehen:

	PLOT	SIN	180
Funktion:	Setze Punkt auf dem Bildschirm	Berechne Sinus	Zahlenwert
Eingaben:	X-Koordinate Y-Koordinate Farbe (optional)	Winkel	--/--
Ausgaben:	--/--	Sinus des Winkels	180

COMMANDS

Es fällt auf, daß PLOT keine Ausgaben macht und 180 keine Eingaben benötigt. Tatsächlich machen alle Wörter, mit denen ein BASIC-Befehl anfangen kann, keine Ausgabe, die von einem anderen Befehlswort als Eingabe verwendet werden kann:

```
SOUND a,b,c,d,e
PLOT x,y
PRINT "hallo"
CLEAR
```

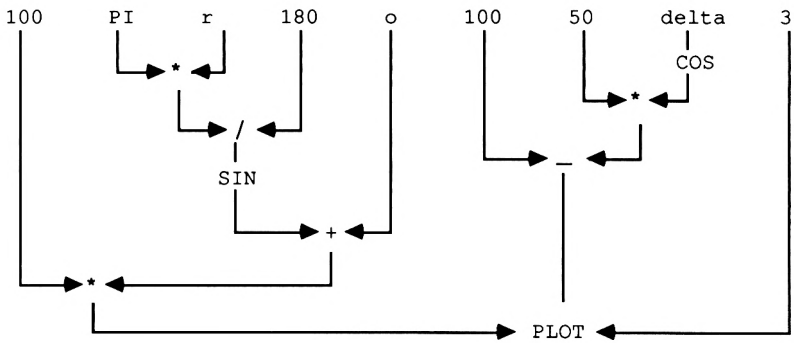
Die Bezeichnung für solche Sprachelemente ist leider nicht ganz einheitlich. Sie sollen aber im folgenden *Command* (Befehl, Kommando) genannt werden. Das Kommando PLOT darf nicht mit dem vollständigen Befehl PLOT x,y verwechselt werden.

Andererseits benötigen die meisten Elemente aber Eingaben. Woher kommen die? Von anderen Wörtern, die eben Ausgaben machen können wie SIN oder 180. Aber auch hier gibt es wieder Unterschiede: Während 180 so etwas wie ein Deckel auf dem Topf ist, benötigt SIN nun seinerseits selbst wieder eine Eingabe.

Man kann sich hier recht gut vorstellen, wie einzelne Daten an verschiedenen Quellen herausströmen, durch verarbeitende Elemente hindurchfließen und schließlich alle in ein gemeinsames Ziel hineinmünden. Dabei gibt es norma-

lerweise beliebig viele Datenquellen, die immer zu einem Ziel hinfließen. Das folgende Diagramm soll die Datenbewegungen in dem komplizierten PLOT-Befehl von oben veranschaulichen:

```
PLOT 100*(SIN(PI*r/180)+o),100-50*COS(delta),3
```



STATEMENTS

Das Kennzeichen eines vollständigen Befehls ist, daß er bezüglich der Parameterübergabe in sich abgeschlossen ist. Er benötigt keine Eingaben von außen, und er macht keine Ausgaben an andere Sprachelemente. Ein solcher in sich abgeschlossener Befehl wird *Statement* (Anweisung) genannt.

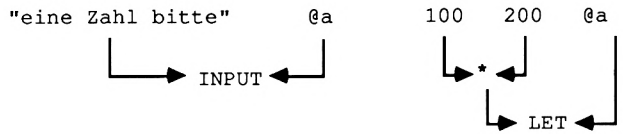
Bei der Definition von Eingabe und Ausgabe in diesem Sinn muß man aufpassen. Hierbei sind nur die Datenwege innerhalb des Statements gemeint, z.B. die Weiterverarbeitung von Ausgaben, die andere Befehlselemente gemacht haben.

Nur um sie muß sich der *Parser* der jeweiligen Programmiersprache kümmern. Der Parser ist die Abteilung, die den Programmtext einliest, untersucht und entsprechende Aktionen veranlaßt. Auch die beiden folgenden Texte stellen korrekt gebildete Statements dar:

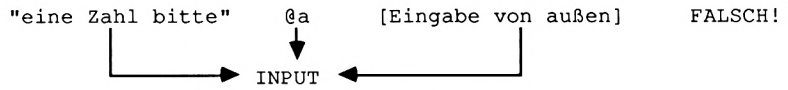
```
INPUT "eine Zahl bitte:",a
```

```
LET b=100*200
```

Hierbei muß der Parser aber ein gewisses Maß an Intelligenz besitzen, da er für die Variablen a und b nicht deren aktuellen Wert, sondern ihre Adresse an die INPUT- bzw. LET-Behandlungsroutine liefern muß. Die INPUT- bzw. LET-Routine benötigt diese Adresse, um den Inhalt der Variablen verändern zu können:

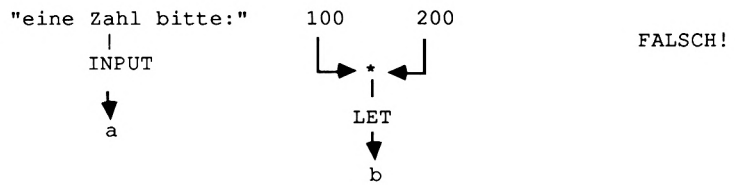


Dabei ist die Vorstellung falsch, daß beispielsweise das Statement mit INPUT von der Eingabe her nicht abgeschlossen sei, weil der Anwender hier ja eine Eingabe machen müßte:



Um diese Eingabe von außen muß sich nämlich nicht der Parser kümmern, sondern das ist die Aufgabe der INPUT-Behandlungsroutine.

Genauso falsch ist die Vorstellung, daß INPUT bzw. LET eine Ausgabe an weitere Programmelemente, nämlich die Variablen, machen würden:



Auch um diese Zuweisung kümmert sich nicht der Parser, sondern auch wieder die Behandlungsroutine des jeweiligen Befehls. Sonst würden ja beispielsweise auch Befehle wie PLOT oder PRINT Ausgaben machen; nur auf dem Bildschirm eben.

Aber diese Ausgaben werden nicht vom Parser an andere Befehlselemente weitergereicht, sondern sind die Aktion, die der Befehl eben bewirken sollte.

SEPARATOREN

Um dem Parser überhaupt zu ermöglichen, die einzelnen Sprachelemente zu erkennen, müssen diese voneinander getrennt werden. So ist das folgende Statement nicht zu entziffern:

```
PLOTSIN(phi),COS(phi)
```

weil zwischen PLOT und SIN kein Leerzeichen steht. Andererseits führt folgender Variablenname nicht zu einer Fehlinterpretation:

```
SINUS=0.5
```

Obwohl SINUS das dem BASIC-Parser bekannte Wort SIN enthält, ist es klar als Variablenname erkennbar, weil nach SIN kein *Separator* kommt.

Alle Befehlselemente müssen durch Trennzeichen voneinander abgegrenzt werden. Das universellste Zeichen ist dabei der *Space*, das Leerzeichen, das nur zum Trennen von Namen usw. dient und keine eigene Bedeutung hat. Leerzeichen können zwischen den einzelnen Befehlszeichen in beliebiger Menge eingefügt werden:

```
PLOT    100- 20*a ,   300-30 * b ,   x    =    PLOT 100-20*a,300-30*b,x
```

Demgegenüber haben die Separatoren *Komma*, *Semikolon*, *Apostroph* usw. zusätzliche Bedeutungen.

Benötigt ein Befehlselement mehrere Eingaben, so müssen diese durch ein Komma getrennt sein. Bei einigen Kommandos gibt es noch Sonderregelungen. So verlangt die Wertzuweisung als Separator das Gleichheitszeichen (=), und im PRINT-Statement können die einzelnen Argumente mit Leerzeichen, Kommas oder Semikolons abgegrenzt werden. Auf das Apostroph folgt bis zum Zeilenende nur noch nicht auszuwertender Text (eine Bemerkung, Remark).

```
LET a=b*2        oder:     PRINT a*20 b-10 c
   ↑ ↑                          ↑    ↑    ↑
```

Auch die einzelnen Statements müssen gegeneinander abgegrenzt werden: Als Grenze gelten dabei der Anfang und das Ende einer BASIC-Zeile und der Doppelpunkt:

```
10  PRINT a:LET a=a+1:GOTO 10
   ↑       ↑         ↑       ↑
```

Einige spezielle Befehlselemente, die durch ein Sonderzeichen dargestellt werden, werden automatisch als ihr eigener Separator erkannt: Plus, Minus, Mal, Geteilt etc.:

```
PRINT 100+200        ist gleichwertig mit        PRINT 100 + 200
        ↑                                                ↑↑↑
```

KLAMMERN

Eine besondere Rolle kommt noch den Klammern zu, die auf jeden Fall auch als Separator wirken.

BASIC unterscheidet ganz wesentlich zwischen den Kommandos und den anderen Sprachelementen, die Ausgaben machen. Benötigt ein Kommando Parameter (Eingaben), so werden diese einfach angehängt. Mehrere Parameter müssen durch einen Separator getrennt werden:

```
SOUND 7,100,100        oder        PRINT a-b,b-c;c-d d-e
      ↑   ↑                                ↑   ↑   ↑
```

FUNKTIONEN, ARGUMENTE

Sprachelemente wie SIN, INSTR oder LOG, die Ausgaben machen, benötigen auch Parameter. Hier müssen sie aber durch Klammern eingeschlossen werden. Das muß geschehen, weil sonst Doppeldeutigkeiten entstehen:

```
PRINT SIN 100 + 200  =  PRINT SIN (100+200)
                   oder PRINT SIN(100) + 200
```

Solche Elemente, die eine Ausgabe liefern, werden als *Funktionen* bezeichnet. Prinzipiell sind zwar auch Funktionen möglich, die mehr als eine Ausgabe machen. Diese sind in BASIC aber von der Sprachstruktur her nicht denkbar. Funktionen haben meist eine Eingabe, wie z.B. SIN(winkel). Das ist aber nicht unbedingt der Fall.

Es gibt auch Funktionen ohne Eingabe oder mit zwei und noch mehr:

```
PRINT RND      oder      PRINT INSTR(5,"1234567","4")
                                    ↑         ↑
```

In diesem Zusammenhang ist es interessant, daß die Boolesche Funktion NOT im Schneider-BASIC ihr Argument *nicht* in Klammern erwartet:

```
PRINT NOT 0 [ENTER]
-1
Ready
```

NOT und "–" (Vorzeichenwechsel) werden deshalb oft als monadische Operatoren bezeichnet.

FELDER, INDIZES

Zweites Einsatzgebiet für Klammern sind die dimensionierten Felder. Das sind strukturierte Variablen, die mehrere Datenplätze umfassen. Um auf ein bestimmtes Datenelement zugreifen zu können, muß noch ein Index oder mehrere Indizes (je nach Dimensionierung) angegeben werden. Diese werden, gerade wie bei Funktionen, von Klammern umschlossen an den Variablennamen angehängt und bei mehreren Indizes durch Kommas getrennt:

	Funktion	Datenfeld	
ein Arg.:	`PRINT SIN(0)`	`PRINT a(7)`	← ein Index
zwei Arg.:	`PRINT TEST(10,0)`	`PRINT b(1,5)`	← zwei Indizes

Weitgehend unbekannt ist, daß die Programmierer des Locomotive BASIC wohl überzeugte Pascal-Anwender waren. Sie haben nämlich eine Pascal-typische Eigenheit ins Schneider-BASIC implementiert: Wie in Pascal kann man in diesem BASIC-Dialekt die Indizes einer Feldvariablen in eckige Klammern einschließen:

```
PRINT a[1,2]
```

Damit ist es rein optisch leichter möglich, Zahlenfelder und Funktionen auseinanderzuhalten.

OPERATOREN, OPERANDEN

Es gibt noch ein drittes Anwendungsgebiet für die runden Klammern: Prioritätssteuerung in arithmetischen Ausdrücken.

Betrachten wir einmal den Ausdruck

```
PRINT 50+60*70 [ENTER]
 4250
Ready
```

Um Formeln in dieser Art in den Computer eingeben zu können, muß der Parser erst einmal den ganzen Ausdruck in seine Elemente zerlegen. Das geht noch vergleichsweise einfach, dafür sind ja die Separatoren da:

```
PRINT 50 + 60 * 70
```

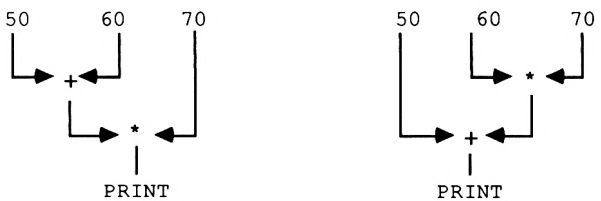
"Plus" und "Mal" sind hier Operatoren. Alle Operationen sind zweiwertige Funktionen. Das heißt, sie benötigen zwei Argumente (die Operanden genannt werden) und liefern einen Funktionswert:

60 * 70 entspricht MULT(60,70)

Nun hat diese Operatoren-Schreibweise einen gewaltigen Haken: Sie ist nicht eindeutig:

`PRINT 50+60*70` = `PRINT (50+60)*70` oder `PRINT 50+(60*70)` ?

Klammern könnten in einem arithmetischen Ausdruck die Rechenreihenfolge festlegen:



Um die Programmierung komplexer Ausdrücke zu vereinfachen, hat man den Operatoren Prioritäten zugeordnet. Ohne eine solche Priorität würde jeder Ausdruck von links nach rechts ausgewertet, wenn keine Klammer etwas anderes befiehlt.

Die Prioritätshierarchie ist folgende:

I)	1.	Potenzierung	^
	2.	Vorzeichenwechsel	–
	3.	Punktrechnung	* und /
	4.	Integer-Division	\
	5.	Restbildung	MOD
	6.	Strichrechnung	+ und –
II)	7.	Vergleich	< > <= >= <> und =

III)	8.	Komplement	NOT
	9.	Und	AND
	10.	Oder	OR
	11.	Exklusiv-Oder	XOR

Es ist interessant festzuhalten, daß Locomotive BASIC das Vorzeichen "-" und die Komplementbildung NOT wie Operatoren behandelt. (Deshalb braucht das Argument von NOT auch nicht in Klammern zu stehen.)

Übrigens ist auch die String-Verknüpfung eine Operation:

```
PRINT a$ + "abc"
```

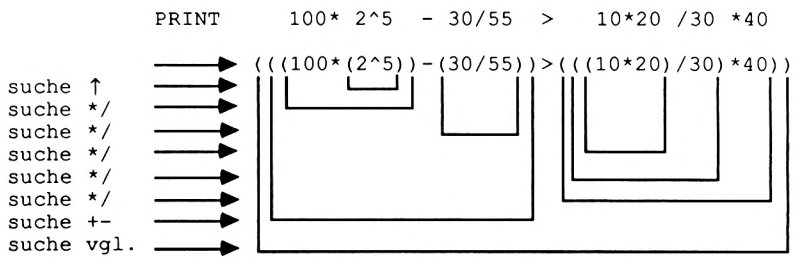
Da es für Strings aber nur diese einzige Operation gibt, braucht der Parser hier keine Prioritäten auszuwerten. Alle Operationen der Gruppe I benötigen als Eingabe einen numerischen Wert und liefern auch einen solchen als Ausgabe.

Die Vergleiche (Gruppe II) benötigen als Eingaben auch numerische Werte, liefern aber Boolesche Ausdrücke als Ausgabe.

Und die Booleschen Operatoren der Gruppe III schließlich erwarten und liefern Boolesche Werte.

I:	numerisch	→ numerisch
II:	numerisch	→ boolean
III:	boolean	→ boolean

Wie geht nun aber ein Parser vor, um die verschiedenen Operationen in der korrekten Reihenfolge zu erledigen? Ein möglicher Weg wäre, zuerst den gesamten Ausdruck in seine Elemente zu zerlegen und dann den Ausdruck, mit der höchsten Priorität nach Operatoren beginnend, zu durchsuchen und diesen jeweils mit seinen Operanden zu klammern:



Allgemein ist ein solcher *Term* (arithmetischer Ausdruck) wie folgt aufgebaut:

- Ein Term ist im einfachsten Fall eine *Zahl*, einfache *Variable* oder *Funktion* ohne Argumente.
- Es kann aber auch eine Variable mit Indizes oder eine Funktion mit Argumenten sein. Diese müssen dann in einer nachstehenden Klammer angegeben werden. Bei mehreren Argumenten bzw. Indizes werden sie durch Kommas getrennt. Diese Argumente/Indizes sind im allgemeinen Fall auch wieder numerische Ausdrücke, also Terme.
- Ein Spezialfall der Funktionen sind die Operationen. Diese Funktionen haben zwei Argumente (Operanden), die selbst auch wieder Terme sind und die durch einen Operator getrennt werden.

Weil ein Term immer wieder auch aus untergeordneten Termen bestehen kann, ist eine Auswertung fast nur mittels Rekursion möglich. Dabei wird ein Stapel benötigt, auf dem die Zwischenergebnisse (Indizes, Parameter oder Operanden) festgehalten werden können.

Syntax und Semantik

Unter der Syntax versteht man die Grammatik einer Sprache. Welche Zeichen- und Ziffernfolgen ein korrektes Element der Sprache bilden, welche Kombinationen von Elementen ein Statement bilden, welche Sprachstrukturen es gibt und wie sie gebildet werden.

Die Semantik beschreibt den Sinninhalt eines Befehls, was sich der Programmierer dabei gedacht hat. Nur syntaktisch korrekt gebildete Befehle sind auch semantisch "sinnvoll". Umgekehrt trifft aber der Schluß "alle syntaktisch korrekten Befehle ergeben auch (semantisch) einen Sinn" nicht zu:

"Alle Neger sind Menschen"

ist sowohl semantisch als auch syntaktisch korrekt. Demgegenüber ist der Satz

"Alle Menschen sind Neger"

zwar syntaktisch richtig, inhaltlich ist er aber falsch. Hier liegt ein Sinn- oder eben semantischer Fehler vor.

Für den Parser einer Programmiersprache ist es vergleichsweise einfach, Syntaxfehler zu erkennen. Deswegen ist diese Fehlermeldung wohl auch die häu-

figste, die der BASIC-Interpreter im Schneider CPC meldet:

```
PRNT a*2 [ENTER]
Syntax error
Ready
```

Semantische Fehler sind meist erst im Programmlauf erkennbar:

```
10 DIM a(10)
20 FOR i=9 TO 99:PRINT a(i):NEXT
RUN [ENTER]
 0
 0
Subscript out of range
Ready
```

In der Regel bereiten Syntaxfehler dem Programmierer keine großen Probleme, weil sie recht schnell vom BASIC-Interpreter selbst erkannt werden.

Auch der Schaden, den semantische Fehler anrichten können, hält sich in BASIC in Grenzen. Wenn man nicht gerade eine Schleife mit POKEs oder CALLs falsch konstruiert, meldet sich BASIC mit einer Fehlermeldung, sobald es kritisch wird.

In bezug auf syntaktische und semantische Fehler gibt es gerade zwischen Interpretern und Compilern große Unterschiede.

Ein Interpreter bemerkt einen Syntaxfehler erst, wenn er den entsprechenden Befehl auch tatsächlich abarbeiten will. Ist er in einem Programmpfad versteckt, der nur ganz selten benutzt wird, so kann es sein, daß ein Programm, das man schon lange fehlerfrei glaubte, plötzlich mit "Syntax error" endet.

Dasselbe gilt natürlich auch für die semantischen Fehler.

Bei Compilern liegt die Sache anders: Der Compiler übersetzt den Programmtext gleich in Maschinensprache, also auch die selten abgearbeiteten Programmpfade. Ein Compiler findet also auf jeden Fall jeden Syntaxfehler im Programm.

Kapitel 2

Das Innenleben der CPC-Rechner

Die CPU Z80

Die CPU (Central Processing Unit) oder auf deutsch der Mikroprozessor ist die Zentrale eines jeden Computers. Frei übersetzt bedeutet CPU etwa: zentrale Prozeß-Steuerungseinheit.

Und das ist sie in der Tat: Von hier aus werden alle anderen ICs programmiert; alle Aktionen im Computer werden von ihr entweder direkt durchgeführt oder doch zumindest veranlaßt. Sie ist das Bauteil im Computer, das ihn so flexibel macht: Sie ist praktisch in beliebig komplexem Umfang programmierbar. Die Leistungsfähigkeit der CPU beeinflußt ganz entscheidend die Fähigkeiten des Gesamtsystems.

Die CPU übernimmt sofort nach dem Einschalten des Rechners die Kontrolle über das gesamte System. Anlegen der Versorgungsspannung und ein kurzer Impuls an einem ihrer Pins (Anschlußstifte); das ist alles, was man machen muß.

Danach ist die CPU aber nicht autark. Sie kann nicht alleine existieren! Nach der Initialisierung (Reset) beginnt sie sofort, das zu machen, was sie bis zum Ausschalten des Computers machen wird: Sie arbeitet ein Programm ab, das nicht in ihr selbst gespeichert ist.

BESCHREIBUNG DES Z80

Der Z80 ist ein 8-Bit-Mikroprozessor. Das bedeutet, daß sein Datenbus genau 8 Bits breit ist. Wie die meisten anderen 8-Bit-Prozessoren auch, verfügt er über einen Adreßbus von doppelter Breite: Mit 16 Adreßbits kann er insgesamt 2^{16} (das sind 65536) verschiedene Speicherzellen adressieren. Mittels der im Schneider CPC angewandten Bank-Switching-Technik allerdings auch noch weit mehr.

Der Z80 verfügt, im Gegensatz zu vielen anderen CPUs, über eine getrennte Ansteuerung von Speicher-ICs und Peripherie-ICs. Damit muß man die verfügbaren Adressen nicht für Speicher und Peripherie aufteilen.

Normalerweise werden Peripheriebausteine nur mit der unteren Hälfte des Adreßbusses adressiert, wodurch sich schon bis zu 256 verschiedene externe Funktionen ansprechen lassen. Mit den im Schneider CPC akzeptierten Einschränkungen ist aber auch der gesamte Adreßbus in voller Breite für I/O-Zwecke verwertbar. Damit wären dann, zusätzlich zu den 65536 Speicherzellen, auch noch 65536 verschiedene Peripheriefunktionen steuerbar.

Zum normalen Interieur jeder CPU gehören ein Programmzeiger (PC), ein Stapelzeiger (SP), ein Akkumulator (Akku oder einfach A) und ein Signalregister (Flags oder F). Das sind alles Register, also Speicherzellen, im Inneren der CPU.

Der Programmzeiger (PC – Program Counter) zeigt dabei jeweils auf den gerade abgearbeiteten Befehl. Das PC-Register ist deshalb wie der Adreßbus 16 Bit breit.

Der Stapelzeiger (SP – Stack Pointer) dient als Zeiger auf die Spitze des Maschinenstapels, auf dem u.a. die Return-Adressen bei Unterprogrammaufrufen abgelegt werden. Wie der PC ist auch dieses Register 16 Bit breit. Der Stack kann daher an jeder beliebigen Stelle im RAM liegen und auch beliebig lang werden.

Der Akku, das Hauptrechenregister, ist nur 8 Bit breit und entspricht somit der Breite des Datenbusses. Die meisten Operationen und Funktionen, die der Z80 durchführen kann, erwarten im Akku ihr bzw. eines ihrer Argumente und legen dort auch das Ergebnis ab.

Das Flag-Register ist zwar auch 8 Bit breit, doch werden hiervon nur 6 Bits auch wirklich benutzt. Davon sind wiederum zwei auszuschließen, die nur für einen einzigen Spezialbefehl, DAA zum Dezimalabgleich beim Rechnen mit BCD-codierten Zahlen, ausgewertet werden. Die verbliebenen vier Bits enthalten das Carry-Flag (Übertrag aus dem 8ten bzw. 16ten Bit), das Zero-Flag (Anzeige, ob ein Ergebnis null ist), das Signum-Flag (Vorzeichen des Ergebnisses – Bit 8 oder 16) und das kombinierte Parity-Overflow-Flag. Hierin wird bei Booleschen Funktionen die Parität der Null- und Eins-Bits im Ergebnis festgehalten (gerade oder ungerade Anzahl) und bei arithmetischen Funktionen ein eventueller Überlauf in das Vorzeichenbit (Bit 8 bzw. 16).

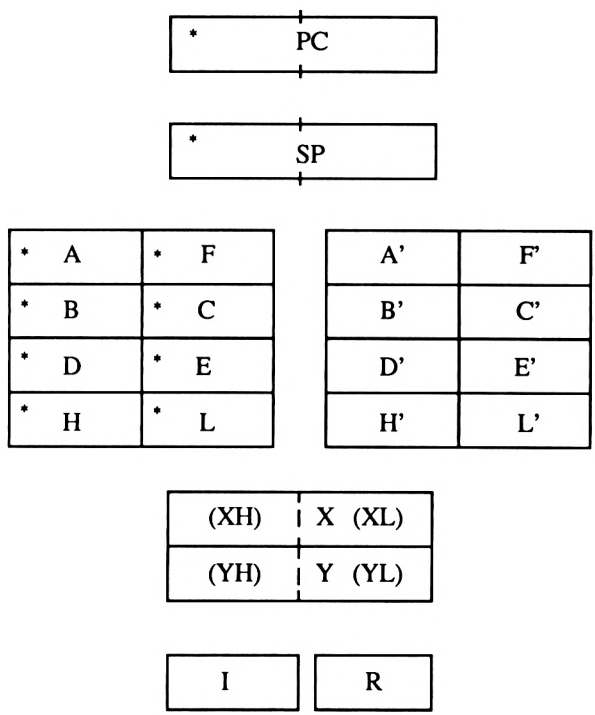
Der Z80 verfügt darüber hinaus noch über viele weitere Register: BC, DE und HL sind 16-Bit-Register, die aber auch getrennt als 8-Bit-B-, C-, D-, E-, H- und L-Register verwendet werden können.

Neben dem Akku erhalten aber auch einige andere Register durch die verfügbaren Befehle eine besondere Bedeutung: So ist das HL-Register das am

vielseitigsten einsetzbare Doppelregister. Bei der 16-Bit-Addition und -Subtraktion fungiert es praktisch als ein 16-Bit-Akku. Außerdem ist die indirekte Adressierung einer Speicherzelle mit dem HL-Register fast gleichwertig mit der Verwendung eines normalen Registers als Datenquelle oder -ziel.

Das B-Register dient als universelles Zählregister für Schleifen.

Folgende Grafik zeigt einen Überblick über die Register des Z80. Jedes Kästchen symbolisiert dabei normalerweise ein 8 Bit breites Register. Einige Register sind mit einem Sternchen * markiert. Diese Register sind auch bei dem Vorgänger des Z80, der CPU 8080 vorhanden:



Das C-Register bildet bei den meisten I/O-Befehlen die Port-Adresse. Im Schneider CPC existiert allerdings eine Besonderheit: Hier wird nur mit einem Nebeneffekt der I/O-Befehle gearbeitet, der das gesamte BC-Register auf den Adreßleitungen A0 bis A15 erscheinen läßt.

Diese Register sind im Z80 jeweils doppelt vorhanden: AF, BC, DE und HL haben jeweils noch eine Kopie, zwischen denen beliebig hin- und hergeschaltet werden kann.

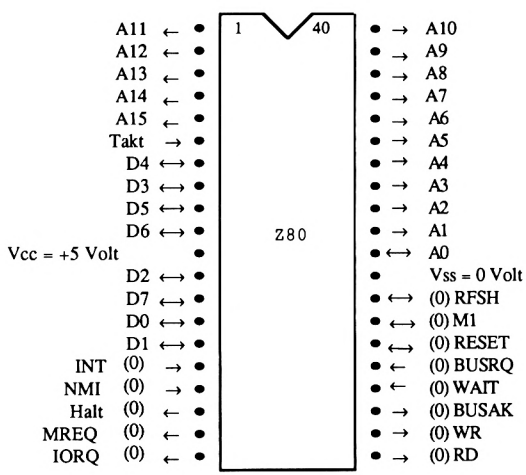
Die 16-Bit-Register IX und IY werden für die indizierte Adressierung verwendet.

Abschließend gibt es noch die Spezialregister I und R. Das 8 Bit breite I-Register wird dabei für einen speziellen Interrupt-Modus als höherwertiges Byte der Adresse einer Tabelle verwendet, in der die Adressen von verschiedenen Interrupt-Behandlungsroutinen stehen können. Dieser Modus wird im Schneider CPC aber nicht verwendet.

Das R-Register ist nur 7 Bit breit und dient als Refresh-Zähler. Der Z80 unterstützt nämlich als eine Besonderheit das Wiederauffrischen dynamischer RAMs. Dazu erzeugt er in der zweiten Hälfte jedes Befehlsholezyklus ein Pseudo-Lesesignal für die Speicherbausteine, wobei er als Adresse auf den untersten 7 Leitungen den Inhalt des R-Registers ausgibt und dieses danach erhöht. Dadurch werden nach 128 (2^7) Befehlen, die der Z80 abarbeitet, die dynamischen RAM-Bausteine vollständig wiederaufgefrischt.

Das I-Register wird im Schneider CPC überhaupt nicht und das R-Register nur im Cassette-Manager zweckentfremdet benutzt.

DIE ANSCHLUSSBELEGUNG DER CPU Z80:



Erklärung zu den Anschlüssen:

Vcc und Vss

Über die Pins Vcc und Vss wird die CPU mit der erforderlichen Betriebsspannung versorgt. Der Z80 benötigt +5V und einen Masseanschluß.

D0 bis D7

Die bidirektionalen Anschlüsse D0 bis D7 bilden den *Datenbus*.

A0 bis A15

Die Pins mit der Bezeichnung A0 bis A15 sind der *Adreßbus*.

Alle noch folgenden Anschlüsse fallen unter die Rubrik *Steuer-* oder auch *Controlbus*:

Takt

Die an Takt anliegende Rechteckspannung kann, je nach verwendeter Serie, eine Frequenz von bis zu 6 MHz haben. Im Schneider CPC wird der verwendete Z80A mit den maximal möglichen 4 MHz betrieben. Die Arbeitsgeschwindigkeit der CPU hängt in erster Linie von der Frequenz dieses Taktes ab. Sie kann nur noch durch Verwendung des Wait-Eingangs gebremst werden.

INT – Interrupt

Der Eingang INT ist Null-aktiv und veranlaßt den Z80, eine Interrupt-Routine abzuarbeiten. Dieser Eingang kann jedoch Software-mäßig mit den Assembler-Befehlen DI und EI gesperrt bzw. wieder zugelassen werden. Er wird außerdem automatisch gesperrt, sobald die CPU die Interrupt-Behandlung aufnimmt.

NMI - Non Maskable Interrupt

Ein Null-Signal am Anschluß NMI bewirkt ebenfalls den Aufruf einer Interrupt-Routine, die aber immer ab Adresse &66 starten muß. Der NMI ist dabei nicht per Software-Befehl abschaltbar. Er kann einem normalen Interrupt dazwischenfunken. Sobald die Behandlung eines NMI aufgenommen wurde, sind weitere Interrupt-Anforderungen (INT oder NMI) gesperrt.

Reset

Dieser Eingang muß nach Einschalten des Computers kurzzeitig auf Null-Pegel gezogen werden, um die CPU zu initialisieren. Dabei werden normale Interrupts gesperrt, wird und der Programmzeiger PC mit &0000 geladen. Die CPU startet mit der Befehlsbearbeitung ab dieser Adresse, sobald die Spannung an diesem Pin auf logisch 1 steigt.

Wait

Mittels dieses Eingangs kann der Z80 asynchron betrieben werden. Bei einem Speicher- oder I/O-Zugriff kann das angesprochene Bauteil diesen Eingang so lange auf Null legen, bis es die Daten bereitgestellt bzw. übernommen hat. Im Schneider CPC wird dieser Eingang jedoch etwas zweckentfremdet gebraucht, um der CPU nur einmal pro Mikrosekunde den Zugriff auf den Systembus zu gestatten. Dazwischen ist immer der CRTC dran, der ständig Daten aus dem Bildwiederholspeicher benötigt.

BUSRQ und BUSAK – Busrequest und Busacknowledge

Mit Hilfe dieser Anschlüsse ist es möglich, mehrere Bausteine am Systembus zu betreiben, die aktiv auf den Bus zugreifen können. Im Schneider CPC werden sie nicht weiter verwendet, sind aber auf den Systembus durchgeführt. Will ein anderes IC auf den Bus zugreifen (beispielsweise ein DMA-Controller oder eine andere CPU) signalisiert sie das dem Z80 durch Null-Legen des Eingangs BUSRQ. Sobald sich der Z80 vom Datenbus zurückgezogen hat (alle Ausgänge werden hochohmig) signalisiert er das duch Null-Pegel von BUSAK. Jetzt kann das andere IC auf den Bus zugreifen. Ist es fertig, nimmt es die Busanforderung an BUSRQ wieder zurück.

Halt

Nach dem Assembler-Befehl Halt geht die CPU in einen Wartezustand über und stoppt die Programmbearbeitung. Um jedoch den Refresh dynamischer RAMs sicherzustellen, arbeitet sie intern ständig NOP-Befehle ab. Das signalisiert sie nach außen hin durch Low-Legen des Halt-Anschlusses. Aus diesem Zustand wird sie durch die nächste Interrupt-Anforderung wieder geweckt: NMI oder auch INT, falls dieser zugelassen ist. Da im Schneider CPC vom NMI-Eingang keinen Gebrauch gemacht wird, kann man die CPU ziemlich sicher aufhängen, indem man folgende zwei Befehle in ein Assembler-Programm einfügt: DI und danach HALT.

RFSH – Refresh

Im Anschluß an jeden M1-Zyklus (Befehlsholezyklus) gibt der Z80, wie bereits erwähnt, eine Refresh-Adresse für dynamische Speicher-ICs aus. Das ist an der Konstellation RFSH und MREQ gleich Null erkennbar.

M1 – Machine Cycle One

Bei den Speicherzugriffen unterscheidet die CPU nach außen hin zwei Arten: normale und M1-Zyklen. Letztere dienen vorzugsweise dazu, den Befehlscode des nächsten Befehls aus dem Speicher zu holen. Das signalisiert sie nach außen hin durch die Kombination M1, RD und MREQ alle gleich Null. Direkt anschließend erfolgt immer ein Refresh.

MREQ – Memory Request

Bei jedem Speicherzugriff wird diese Signalleitung auf Null-Potential gelegt.

IORQ – Input/Output Request

Im Gegensatz dazu signalisiert eine logische Null an diesem Pin, daß die CPU diesmal auf Peripheriebausteine zugreifen will.

RD und WR - Read und Write

Zusätzlich zu MREQ oder IORQ wird auch noch RD oder WR aktiv. Damit wird die gewünschte Transfer-Richtung festgelegt: Bei Null-Potential an RD will die CPU Daten lesen, bei WR will sie Daten schreiben.

ADRESSIERUNGSARTEN DES Z80

Bei den Adressierungsarten wird der Maschinencode-Programmierer beim Z80 nicht gerade verwöhnt. Eine genaue Unterscheidung der verschiedenen Adressierungsarten krankt jedoch immer daran, daß meist mehrere kombiniert angewendet werden. Außerdem kann man, je nach Standpunkt, einen Befehl in die eine oder andere Gruppe einordnen. Und man kann ohne weiteres einzelne Adressierungsarten nach Belieben zu Gruppen zusammenfassen.

Trotzdem wollen wir im folgenden beschreiben, auf welche Arten der Z80 adressieren kann.

Implizit

Dazu gehören alle Befehle, bei denen Datenquelle oder -ziel implizit im Befehlsbyte angegeben sind. Das sind beim Z80 ausschließlich Register. In dem Ein-Byte-Befehl "ADC A,B" wird als Datenziel und als eine Datenquelle implizit das A-Register angegeben. Im weiteren Sinne implizit (laut Zilog "Register direkt") wird das B-Register als die andere Datenquelle festgelegt.

Unmittelbar

Diese Adressierungsart kommt nur als Datenquelle in Frage. Hierbei wird dem Befehlsbyte eine Konstante angehängt, je nach benötigter Länge ein Byte oder ein Word. In "ADC A,120" wird die Konstante 120, ein Byte, unmittelbar angegeben.

Absolut

Datenquelle oder -ziel ist eine oder sind zwei aufeinanderfolgende Speicherzellen. Die Adresse wird dem Befehl in einem Word unmittelbar angehängt. Beispiele sind: "LD_HL,(#8000)", "JP #9000" oder "LD (#FFFF),A".

Zero-Page

Der Z80 verfügt über keine spezielle kurze Adressierung von Speicherzellen in der untersten Speicherseite, also bei Adressen, die sich mit einem Byte darstellen lassen. Er verfügt nur über eine sogenannte modifizierte Seite-Null-Adressierung bei den Restart-Vektoren. Das sind Unterprogramm-Aufrufe, die nur aus einem Byte bestehen und dort implizit (in einem 3-Bit-Adreßfeld) 8 verschiedene Einsprungstellen von 0 bis 56 codiert haben.

Relativ

Diese Adressierungsart ist im Z80 nur für kurze Sprünge implementiert. Dabei wird der Programmzeiger PC um die unmittelbar angegebene Adreßdistanz erhöht oder erniedrigt. Dieser Offset ist dabei ein Byte, das bei gesetztem 7. Bit als eine negative Zahl in Komplementdarstellung interpretiert wird.

Register-indirekt

Als Datenziel oder -quelle dient eine Speicherzelle. Die Adresse ist in einem implizit angegebenen Register enthalten. Außerdem ist die Register-indirekte Adressierung für einen einzigen Sprungbefehl implementiert: 'JP_(HL)' bzw. indirekt zu IX oder IY.

Indiziert

Das ist eine Kombination aus Register-indirekter und relativer Adressierung. Das verwendete Register (IX oder IY) wird implizit angegeben (oder auch unmittelbar). Nach dem Befehlsbyte wird die Adreßdistanz (der Index) unmittelbar angegeben. Dieses Byte wird wie bei der relativen Adressierung zum Register addiert.

Register-indirekt mit Postinkrement oder Predekrement

Alle Befehle, die eine Adresse oder ein Doppelregister auf den Maschinenstapel "pushen" oder von diesem "poppen", verwenden diese Adressierungsart. Hierbei ist Datenziel (beim Pushen) bzw. Datenquelle (beim Poppen) eine Speicherzelle, die durch das SP-Register, also durch den Maschinenstapelzeiger adressiert wird. Das SP-Register taucht im Mnemonic nicht auf, trotzdem verwenden u.a. folgende Befehle diese Adressierungsart: "PUSH HL", "POP IX", "CALL #BC00", "RET" oder "RST 0".

Alle pushenden Befehle arbeiten mit Predekrement. Das heißt, bevor ein Byte SP-Register-indirekt abgespeichert wird, wird dieses erniedrigt. Die poppenden Befehle sind Postinkrement. Nach jedem SP-Register-indirekten Lesen einer Speicherzelle wird das SP-Register inkrementiert. SP-indirekte Adressierung ist nur mit Words möglich.

DATENBREITE

Eine andere Möglichkeit, den Datenzugriff des Z80 zu unterteilen, ist die Betrachtung der betroffenen Daten selbst. Hier ist der Z80 recht komfortabel mit 4 verschiedenen Möglichkeiten ausgestattet:

Bytes

Meistens wird mit Bytes gearbeitet, da der Z80 als 8-Bit-CPU auch einen 8 Bit breiten Datenbus hat. Für Bytes werden deshalb auch die meisten Adressierungsarten zur Verfügung gestellt.

Words

Die nächste Gruppe stellen die 2-Byte-Befehle dar, die mit ein Grund dafür sind, daß der Z80 bis heute erfolgreich überlebt hat. Obwohl er nur ein 8-Bit-Prozessor ist, kann er bereits mit 16 Bit breiten Worten arbeiten! Aller-

dings gibt es hier Einschränkungen bei den arithmetischen Operationen: Boolesche gibt es keine und auch bei den Adressierungsarten sind einige nicht implementiert. Andererseits arbeiten die Stack-Befehle nur mit Doppelregistern.

Bits

Die dritte Gruppe sind die Bit-Setz- und Testbefehle. Hier sind praktisch alle Adressierungsarten erlaubt, die auch bei den Bytes zulässig sind. Nur die unmittelbare Adressierung wird durch eine implizite ersetzt: Will man Register A mit 0 laden, muß man das Byte 0 unmittelbar an das Befehlsbyte anhängen: "LD A,0". Soll aber nur ein Bit in A mit 0 geladen werden, wird hierfür ein eigener Befehl verwendet: "RES 3,A" setzt Bit 3 von Register A auf 0 zurück.

Nibbles

Bei dieser Möglichkeit gibt es nur die Nibble-Tauschbefehle RLD und RRD, die implizit die untere Hälfte des A-Registers und Register-indirekt zu HL eine Speicherzelle als Datenziel und -quelle benutzen.

DIE VERSCHIEDENEN INTERRUPT-MODI DES Z80

Der Z80 kann in drei verschiedenen Interrupt-Modi betrieben werden. Wir wollen dafür die übliche Numerierung von 0 bis 2 verwenden.

Die verschiedenen Modi beziehen sich ausschließlich auf den normalen Interrupt, der NMI führt immer nur einen Sprung zur Adresse &66 aus. Die normalen Interrupts lassen sich immer, unabhängig vom gewählten Modus, mittels der Assembler-Befehle EI und DI verbieten und wieder zulassen.

Sobald ein Peripheriebaustein einen Interrupt auslöst, bearbeitet die CPU den Befehl, an dem sie gerade ist, zu Ende und beginnt dann mit einem speziellen Interrupt-Acknowledge-Zyklus. Das ist ein Buszugriff, der sich von den normalen Speicher- und I/O-Zugriffen leicht unterscheiden läßt: Zu einem bestimmten Zeitpunkt werden nämlich die Ausgänge M1 und IORQ zusammen aktiv. In diesem Moment kann das IC, das den Interrupt ausgelöst hat, ein Datenbyte auf dem Datenbus ausgeben, das die CPU in ihr Eingangsregister latcht. Je nach eingestelltem Interrupt-Modus wird dieses Datenbyte unterschiedlich interpretiert.

IM0 – Interrupt-Modus 0

Das Datenbyte wird als 1-Byte-Befehl behandelt und ausgeführt. Sinnvoll ist hier nur der Einsatz eines Restart-Befehls, wodurch bis zu acht verschiedene Interrupt-Behandlungsroutinen unterschieden werden können.

IM1 – Interrupt-Modus 1

Dieser Modus wird im Schneider CPC angewandt. Bei diesem Modus ignoriert die CPU einfach das Datenbyte und führt selbständig einen Restart zur Adresse 56 aus.

IM2 – Interrupt-Modus 2

Dieser Modus ist der mächtigste von allen dreien. Hierbei können bis zu 128 Routinen unterschieden werden. Der Z80 nimmt hierbei das Datenbyte vom Peripheriegerät als niederwertiges Byte und das I-Register als höherwertiges Byte und bildet daraus eine Adresse. An dieser Stelle holt er aus dem Speicher zwei Bytes, die er nun als Adresse der Interrupt-Behandlungsroutine interpretiert und aufruft. Mit dem I-Register legt man also die Basis einer Tabelle mit Routinenadressen fest, und das Peripheriegerät sucht sich dann einen Eintrag aus der Tabelle aus.

Da der Z80 im Schneider CPC im Interrupt-Modus 1 betrieben wird, hat das I-Register hier keine Aufgabe zu erfüllen. Es wird auch vom Betriebssystem nicht für irgendwelche anderen Zwecke gebraucht. Wer also z.B. einen Zwischenspeicher für ein Byte in einem Maschinencode-Programm sucht, findet hier noch ein sicheres Plätzchen.

DAS REFRESH-REGISTER

Ähnlich verhält es sich beim R- oder Refresh-Register. Die dynamischen Speicher im Schneider CPC werden nicht durch die CPU wiederaufgefrischt. Das besorgt der Video-Controller, indem er 50 bzw. 60mal pro Sekunde (je nach Fernsehnorm) den Bildwiederholspeicher ausliest. Das Refresh-Signal der CPU wird nicht zu den Speicher-ICs weitergeleitet, steht aber an der Rückseite des Computers am Systembus-Anschluß zur Verfügung.

Anders als beim I-Register wird das Refresh-Register aber mit jedem M1-Zyklus (und damit mit jedem Refresh) um eins erhöht. Ausgenommen davon ist nur das oberste Bit: Bit 7 bleibt immer auf dem Wert stehen, mit dem man es programmiert hat. Diesen Effekt macht man sich im Schneider CPC bei den Kassetten-Schreib- und -Leseroutinen zunutze, wo man ja für Timing-Zwecke

einen genauen Zähler braucht. Normalerweise benutzt man dafür ein normales Register, meist das B-Register, weil das bei dem relativen bedingten Sprung "DJNZ dis" automatisch erniedrigt wird. Hierfür das Refresh-Register zu benutzen, ist zumindest eine originelle Idee.

Ein weiteres, möglicherweise sinnvolleres Einsatzgebiet dieses Registers ist eine Verwendung als Zufallszahlen-Generator, allerdings nur im Bereich 1 bis 127. Man muß allerdings darauf achten, daß zwischen zwei Lesezugriffen auf dieses Register nicht eine konstante Anzahl von Befehlen abgearbeitet wird, sonst haben zwei aufeinanderfolgend gelesene "Zufallszahlen" immer den gleichen Abstand zueinander. Im allgemeinen wird das aber schon allein durch den Interrupt, der ja regelmäßig dazwischen auftritt, garantiert.

DER ZWEITE REGISTERSATZ

Bei der Aufzählung der vielen Register im Z80 wurde bereits beschrieben, daß es für die wichtigsten Registerpaare AF, BC, DE und HL noch jeweils eine Kopie gibt, zwischen denen man willkürlich hin- und herschalten kann. Das Umschalten geschieht dabei jedoch in Gruppen:

Mit dem Befehl "EX AF,A'F'" werden der Akku und das Flag-Register mit ihrem Zweitregister vertauscht.

Der Befehl "EXX" besorgt das für BC, DE und HL zusammen. Diese Register können nicht einzeln umgeschaltet werden.

Normalerweise sind die beiden Registersätze vollkommen gleichwertig. Ein Programm kann beliebig zwischen dem einen und dem anderen Satz hin- und herschalten. Streng genommen ist auch die Unterscheidung in einen ersten und in einen zweiten Registersatz nicht ganz korrekt. Beide Sätze sind für die CPU vollkommen gleichwertig.

Eine Wertung der beiden Sätze ergibt sich erst durch das Programm. So ist es bei vielen Betriebssystemen üblich, nur mit einem Satz zu arbeiten, der dann zum "ersten" Satz wird. Der "zweite" Registersatz wird ausschließlich für die Interrupt-Routine(n) reserviert. Da diese das Hauptprogramm ja jederzeit unterbrechen können, dürfen sie keine Register, die das Hauptprogramm benutzt, verändern.

Normalerweise pusht eine Interrupt-Routine zuerst einmal alle Register, die es benötigt, auf den Maschinenstapel, arbeitet dann seine Routinen ab und restauriert die Registerinhalte anschließend wieder, indem es sie vom Stapel zurück-

holt. Beschränkt sich das Hauptprogramm aber auf einen Registersatz, kann man diesen viel schneller retten, indem man einfach alle Register austauscht: "EX AF,A'F'" und "EXX".

Beim Schneider CPC wird hier ein gemischtes System betrieben. Auf jeden Fall kann man hier auch nicht ohne besondere Vorkehrungen auf den zweiten Registersatz zugreifen. Eine Unterteilung in einen "ersten" und "zweiten" Satz ist also durchaus gerechtfertigt.

UNTERPROGRAMMAUFRUFE

Unterprogramme werden beim Z80 mit dem Assembler-Befehl CALL oder RST aufgerufen. Bei CALL muß man die gewünschte Adresse unmittelbar angeben. Außerdem kann man bei Bedarf implizit im Befehlsbyte eins der 4 Flags testen, um nur bei erfüllter Bedingung zu verzweigen. RST ist das Mnemonic für die Restarts, bei denen implizit im Befehlsbyte eine der 8 möglichen Einsprungadressen codiert ist. Eine bedingte Verzweigung ist bei ihnen nicht möglich.

Bei den Restarts gibt es in verschiedenen Assemblern zwei unterschiedliche Methoden, mit denen man angeben kann, welche Adresse man aufrufen will: Einige Assembler benutzen hierfür die Einsprungadressen (0, 8, 16, ..., 56), andere numerieren sie einfach durch (0, 1, ..., 7). Glücklicherweise gibt es dadurch aber keine größeren Probleme, da es zwischen beiden Methoden keine Überschneidungen gibt (außer der Null, die aber in beiden Fällen denselben Restart bezeichnet).

Der dazu passende Abschlußbefehl ist RET, mit dem das gerufene Unterprogramm wieder zum Hauptprogramm zurückkehrt. Streng genommen ist auch das eine Verzweigung. Auch RET kann implizit an eine Bedingung geknüpft sein.

Da das Unterprogramm mit RET wieder an die Stelle des Aufrufs zurückkehren soll, muß beim Aufruf der Programmzeiger PC nicht nur auf die Adresse des Unterprogramms eingestellt, sondern dessen aktueller Inhalt vorher erst gerettet werden. Ein einfacher Sprung läßt sich wie folgt beschreiben:

```
JP #9000      entspricht      LD PC,#9000
```

Ein Unterprogramm-Aufruf ist dagegen eine Befehlskombination:

```
CALL #9000    entspricht      PUSH PC
                              LD   PC,#9000
```

Und der Abschluß eines Unterprogramms:

```
RET          entspricht          POP PC
```

Für einen Unterprogramm-Aufruf wird also immer die Rücksprungadresse auf den Maschinenstapel gepusht. Diese Adresse zeigt dabei immer auf den Beginn des nächsten Befehls.

Auf dem Maschinenstapel ist die Rücksprungadresse dann ungeschützt für das Unterprogramm erreichbar. Falls dieses hier einige Manipulationen vornehmen will (was im Betriebssystem des Schneider CPC ausgesprochen häufig der Fall ist), braucht es sie nur mit "POP register" vom Stapel wieder runterzuholen.

Es ist aber auch der umgekehrte Fall denkbar: Ein Doppelregister wird auf den Stack gepusht und dann mittels RET in den Programmzeiger PC geladen. Der Effekt ist ein indirekter Sprung zu der Adresse, die in dem Doppelregister angegeben war:

```
PUSH BC
RET
```

entspricht dem nicht existierenden Z80-Befehl

```
JP (BC)
```

16-BIT-WORTE SPEICHERN

Die verschiedenen CPU-Hersteller haben sich immer noch nicht einigen können, in welcher Reihenfolge sie die beiden Bytes eines 16-Bit-Wortes im Speicher ablegen sollen: Das höherwertige Byte zuoberst und das niederwertige in der Adresse darunter oder umgekehrt.

Der Z80 verwendet die "umgekehrte" Version: Zuerst kommt das niederwertige Byte (in der niedrigeren Adresse) und darüber das höherwertige. Es ergibt sich also folgende Entsprechung:

```
DEFW #1234   entspricht DEFB #34        ' niederwertig
                        DEFB #12        ' höherwertig
```

oder:

Adresse a	enthält #34
Adresse a+1	enthält #12

Diese Reihenfolge wird vom Z80 immer und ohne Ausnahme eingehalten.

BESONDERHEITEN DES Z80 IM SCHNEIDER CPC

Eine Besonderheit des Z80 beim Schneider CPC wurde ja schon erwähnt: das Refresh-Register. Auf eine andere Eigenheit, die verschiedenen Speicherbänke, wollen wir nun näher eingehen:

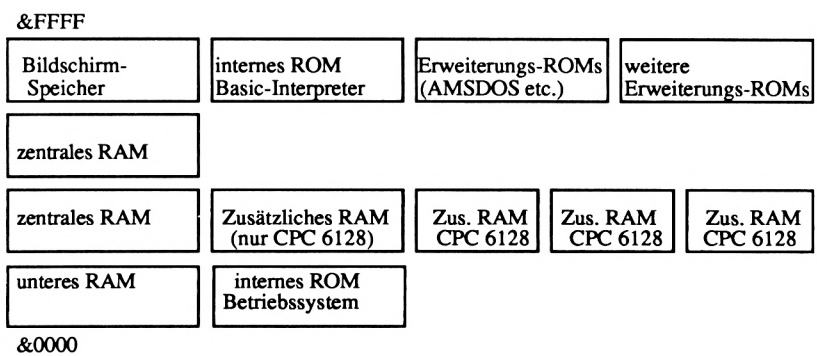
Bankswitching

Ein echtes Handicap für 8-Bit-Mikroprozessoren ist ihr vergleichsweise geringer Adreßumfang. Mit 16 Adreßleitungen kann man eben nur 2^{16}, also 65536, verschiedene Speicherplätze zu je 8 Bit adressieren. Und damit müssen Betriebssystem, BASIC-Interpreter, BASIC-Programm, Daten- und Bildschirmspeicher auskommen. Und da der Schneider CPC auch als Spiele-Computer genutzt wird, hat er prinzipiell einen grafikorientierten Bildschirmspeicher, daher geht allein dafür schon ein Viertel des adressierbaren Speichers weg.

Hätte man sich bei AMSTRAD nicht für das Speicher-Banking entschieden, so müßte man ganz massive Abstriche an BASIC und Betriebssystem machen. Es würden dem Anwender sonst nur noch knappe 16000 Bytes für eigene Programme und Daten zur Verfügung stehen!

Die Hardware ist so konzipiert, daß man einzelne Speicherbereiche auf Eis legen kann. Auch wenn die Z80 die passende Adresse ausgibt, fühlen sie sich nicht mehr angesprochen. Hierdurch kann man dann mehrere Speicherzellen mit denselben Adressen ansprechen. Im Schneider CPC umfassen alle diese Blocks grundsätzlich ein ganzes Speicherviertel. Eine Bank-Auswahl bezieht sich also immer nur auf die obersten beiden Adreßbits A14 und A15.

Der Gesamtspeicher des Schneider CPC ist dabei wie folgt unterteilt:



Dabei werden alle Erweiterungs-ROMs, die man am Systembus des Schneider CPC hinten ansteckt, immer im oberen Speicherviertel eingeblendet. Eine genauere Untersuchung des Speichers im CPC folgt bei der Behandlung der ULA und des PAL-Bausteins.

Das Umschalten der Speicherbänke (Bankswitching) gestaltet sich aber nicht ganz einfach. So kann ein Programm, das sich in einer bestimmten Speicherbank befindet, diese Bank nicht selbst ausblenden (beispielsweise, wenn eine Routine im Betriebssystem aus dem unteren RAM lesen will). Sie würde sich ja selbst das Programm unter den Füßen wegziehen.

Restart-Vektoren im Schneider CPC

Aus diesem Grund hat man nun beim Schneider CPC die Restart-Vektoren etwas zweckentfremdet. Mit ihnen werden jetzt neue Befehle simuliert, die der Z80 nicht kennt.

Der Restart 1 (LOW JUMP) führt einen Sprung ins untere ROM aus. Dazu wird das untere ROM eingeblendet und die Adresse für die gewünschte Routine geholt und angesprungen. Diese Adresse wird dabei "unmittelbar" angegeben. In einen Assembler eingegeben, sieht das dann so aus:

```
RST  1
DEFW adresse
```

Der Restart 1 simuliert dabei den Z80-Befehl "JP adresse", nur daß vorher das untere ROM eingeblendet wird. Der Restart trifft sogar noch Vorbereitungen, daß nach dem RET der so gerufenen Routine die alte Speicherkonfiguration wieder so hergestellt wird, wie sie es vor dem Aufruf war. Man kann also mit diesem neuen Befehl aus dem unteren RAM eine Routine im unteren ROM aufrufen, was sonst nicht ohne Umweg über das zentrale RAM möglich wäre. Das ist es natürlich auch so nicht. Die Behandlungsroutine für RST 1 liegt deshalb auch im zentralen RAM.

Ähnlich funktionieren auch die Restarts 2, 3 und 5, wobei jedoch 2 und 3 einen Unterprogramm-Aufruf ersetzen.

Eine weitere Funktion ist mit dem RST 4 implementiert: Dieser ersetzt den Z80-Befehl "LD A,(HL)", wobei immer aus dem RAM gelesen wird. Die umgekehrte Funktion "LD (HL),A" ist nicht nötig, da alle Speicher-Schreibbefehle immer an das eingebaute RAM gehen, unabhängig davon, ob an der entsprechenden Stelle gerade ein ROM eingeblendet ist oder nicht.

Das ist andererseits jedoch recht unglücklich, da man den CPC aus diesem Grund nicht so ohne weiteres um ein paar RAM-Bänke erweitern kann. Diese würden ja wie die Erweiterungs-ROMs im oberen Speicherviertel eingeblendet. Man kann sie zwar durchaus beschreiben und wieder auslesen, die Schreibbefehle gingen aber automatisch auch an das eingebaute RAM, und da befindet sich normalerweise der Bildschirm.

Da die Restarts unabhängig von der aktuellen Speicherkonfiguration erreichbar sein müssen, ist der unterste Bereich des Kernel-ROMs bis Adresse 63 in das RAM kopiert. ROM und RAM sind hier also identisch.

Non Maskable Interrupt

Obwohl dieser Eingang am Systembus durchgeführt ist, kann man den NMI im Schneider CPC normalerweise nicht nutzen. Der NMI wird nicht benutzt, da es einige Routinen gibt, bei denen ein Interrupt das Timing empfindlich stören würde. Solche Stellen sind beispielsweise die Kassetten- und Disketten-Schreib- und -Leseoperationen und die Programmierung des Sound-Chips.

Wer ihn dennoch benutzen will, muß sich eine eigene Behandlungsroutine im RAM schreiben. Dann muß er aber auch sicherstellen, daß das untere Kernel-ROM nie eingeblendet wird, weil man die Änderung ja schlecht auch im ROM vornehmen kann. Etwas genauer ausgedrückt: Man muß sich das Betriebssystem komplett neu schreiben und im RAM ablegen. Alternativ dazu besteht nur die Möglichkeit, das neue Betriebssystem in ein EPROM zu brennen und mit dem ROM im Computer auszutauschen oder, mit einer entsprechenden Schaltungslogik, hinten am Systembus anzustecken (das geht, man kann von außen auch das untere ROM ausblenden).

Normaler Interrupt

Der Z80 wird im Schneider CPC im Interrupt-Modus 1 betrieben. Aus diesem Grund führt er mit jedem Interrupt-Signal (300mal pro Sekunde) einen Restart 7 durch. An der entsprechenden Adresse 56 (im ROM und im RAM identisch!) steht jedoch nur ein Sprung zu der eigentlichen Routine im ungebankten zentralen RAM.

Hier wird mittels eines Tricks der normale intern von der ULA erzeugte Interrupt von externen Interrupt-Anforderungen von Geräten am Systembus unterschieden. Dieser Trick ist verantwortlich dafür, daß man das AF-Registerpaar aus dem zweiten Registersatz normalerweise nicht benutzen kann.

Externe Interrupt-Quellen dürfen ihr Interrupt-Signal erst zurücknehmen, wenn ihre zugehörige Treibersoftware sie dazu explizit auffordert. Der interne Interrupt wird dagegen sofort zurückgenommen, sobald die CPU die Interrupt-Behandlung aufgenommen hat. Das wird in der normalen Interrupt-Routine getestet, indem in der Routine selbst eine Interrupt-Behandlung wieder zugelassen wird. Steht das Interrupt-Signal noch an, ist es ein externer Interrupt, und in der Interrupt-Routine erfolgt eine weitere Unterbrechung, die dann die Behandlungsroutine für den externen Interrupt aufruft.

Dieser Trick macht nun allerdings den Entwicklern von Zusatz-Hardware das Leben schwer: Die Standard-Peripheriebausteine für den Z80 haben nämlich alle dasselbe Verhalten wie der interne Ticker-Interrupt.

Anschließend werden in der Interrupt-Routine die Register BC, DE und HL gegen ihre Pendants im zweiten Registersatz vertauscht. Im B'C'-Register ist dabei immer der passende Wert gespeichert, um mit OUT (C),C die momentane Speicherkonfiguration wieder herzustellen. Das wird in der Interrupt-Routine ausgenutzt, da diese auch das untere ROM einblenden muß, um hier einige Routinen abzuarbeiten.

Busverbindung im (Mikro-) Sekundentakt

Die größte Eigenart, zu der man den Z80 im Schneider CPC gezwungen hat, ist aber die Verwendung des WAIT-Eingangs der CPU. Normalerweise gibt die- ser Eingang langsamen Speichern und langsamen Peripheriegeräten die Mög- lichkeit, die CPU auf ihr gemächliches Tempo herunterzubremsen. Solange sie ihre Daten noch nicht bereit haben, legen sie diesen Eingang einfach auf Null-Potential, wodurch die CPU einfach mitten im Befehl so lange Warte- takte einlegt, bis der WAIT-Eingang wieder positiv wird.

Im CPC erfüllt er diese Aufgabe zwar auch noch, und er ist auch zum Systembus-Stecker durchgeführt. Die Hauptaufgabe dieses Eingangs ist es jedoch, der CPU nur genau einmal in jeder Mikrosekunde einen Zugriff auf die Speicher-ICs zu erlauben. Die Speicher-Schreib- und -Ladezyklen des Z80 können ja (solange vom WAIT-Eingang kein Gebrauch gemacht wird) 3 oder 4 Taktperioden lang sein. Zusätzlich greift der Z80 im Befehlsholzyklus (M1) sogar zweimal zu: einmal, um ein Befehlsbyte aus dem Speicher zu holen und direkt danach die Refresh-Adresse für die dynamischen RAMs.

Im Schneider CPC ist das aber anders. Die Adressen zu den eingebauten dynamischen RAMs sind über Multiplexer geführt, mit denen sie zwischen den Adreßleitungen von der CPU und vom Video-Controller umgeschaltet werden können. Dieses Umschalten übernimmt das Gate Array. Da der Video-Con-

troller pro Mikrosekunde zwei Bytes aus dem Bildwiederholspeicher benötigt und dabei unbedingten Vorrang vor der CPU hat (die Punkte im Monitorbild sollen ja nicht hin- und hertanzen) gibt er den Takt an: zwei Bytes für den CRTC und dann, ganz kurz, auch eins für die CPU. Um dem Z80 dabei diesen Takt aufzuzwingen, wird er mittels WAIT-Eingang synchronisiert: Drei Takte lang wird dieser Eingang vom Gate Array auf Null-Potential gezogen, nur jeden vierten Takt ist die CPU wieder aktiv. Da der Eingangstakt für den Z80 genau vier MHz beträgt, dauern 3+1=4 Takte genau eine Mikrosekunde.

Die Verarbeitungsgeschwindigkeit des Z80 leidet kaum unter diesen Bedingungen: Viele Buszugriffe (M1 und IORQ) dauern sowieso vier Takte und werden gar nicht beeinflußt. Der Z80 testet den WAIT-Eingang nur einmal pro Buszugriff. War ein Zyklus genau 4 Takte lang, fällt der nächste WAIT-Test genau wieder in die Lücke. Die restlichen 3-Takt-Zugriffe werden lediglich um einen weiteren Takt gestreckt. Grob überschlagen ergibt sich für den Z80 dadurch eine effektive Taktfrequenz von 3,3 MHz.

Bei zeitkritischen Aufgaben (Kassetten-Lade- und -Schreiboperationen beispielsweise) muß man aber wissen, wieviel Zeit die einzelnen Befehle nun wirklich für ihre Abarbeitung benötigen. Dafür sind die Tabellen in der einschlägigen Literatur zum Z80 nicht mehr zu gebrauchen. Im Anhang dieses Buches sind deshalb die beim Schneider CPC gültigen Zeiten zusammengestellt.

Peripherie-Adressierung

Die letzte ungewöhnliche Behandlung, die sich der Z80 im Schneider CPC gefallen lassen mußte, wurde auch bereits erwähnt: Normalerweise adressiert der Z80 die Peripheriebausteine nur mit der unteren Hälfte des Adreßbusses. Bei den meisten I/O-Befehlen ist jedoch auch bekannt, was dabei auf der oberen Hälfte ausgegeben wird. So wird beispielsweise bei

```
OUT (nn),A  und  IN A,(nn)
```

nicht nur die angegebene Adresse nn auf A0 bis A7 ausgegeben, sondern auch das Register A auf der oberen Adreßhälfte. Noch interessanter wird es bei den I/O-Befehlen, die das C-Register als Port-Adresse benutzen:

```
OUT (C),reg  und  IN reg,(C)
```

Hierbei wird ein beliebiges Register über den I/O-Port BC eingelesen bzw. ausgegeben. Also: Das C-Register liegt wie angegeben auf A0 bis A7 an und das B-Register auf A8 bis A15.

Im Schneider CPC wird nun ausschließlich mit diesen beiden Befehlen gearbeitet. Dabei interessiert noch nicht einmal, daß das C-Register auf der unteren Adreßhälfte liegt, benutzt wird (mit einer Einschränkung) ausschließlich das obere Byte! Wenn man also im CPC-Betriebssystem ständig auf den Befehl "OUT (C),C" trifft, muß man sich vor Augen halten, daß der benutzte Effekt eher einem (nicht existierenden) Befehl "OUT (B),C" entspricht.

Wieso man bei AMSTRAD auf diese Idee verfallen ist, ist nicht ganz klar. Sicher ist zumindest, daß man so eine volle 16-Bit-Adresse auch für die I/O-Programmierung schaffen wollte. Es ist nämlich eine ausgesprochen praktische Unart, beim Z80 die Port-Adressen nicht vollständig auszudecodieren.

Jedes Adreßbit wird einfach einem bestimmten Peripheriebaustein zugeordnet. Normalerweise sind bei einem I/O-Zugriff alle Adreßbits auf 1 gesetzt. Nur das Bit für den Baustein, den man ansprechen will, wird auf 0 gesetzt. Dadurch spart man sich eine ganze Menge an Decodierlogik: Man braucht ja nur noch das IORQ-Signal und das entsprechende Adreßbit zu verknüpfen und erhält direkt ein Chip-Select-Signal für das IC.

Interessant wird es aber, wenn man sich die verwendeten Adreßbits anschaut: *Alle* im CPC verwendeten Bausteine werden mit Adreßleitungen aus dem oberen Byte angesprochen.

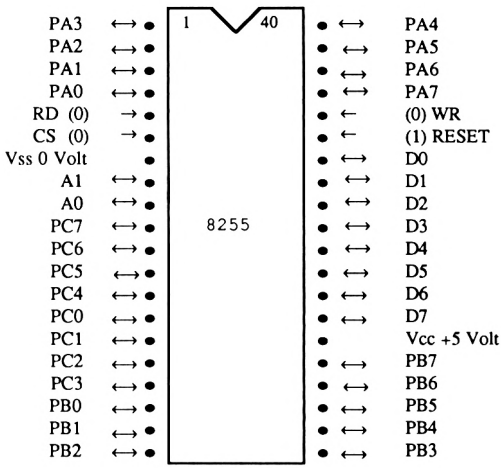
Das untere Adreßbyte ist (bis auf eine Ausnahme) immer völlig ohne Belang. Da hätte man ebensogut auch beim unteren Adreßbyte und damit beim Standardverhalten des Z80 bleiben können. So hat man sich aber unter anderem die Möglichkeit von Block-I/O-Transfers verbaut: Diese benutzen nämlich das C-Register als Port-Adresse und das B-Register als Zähler. Und daß man dann mit dem B-Register nichts auf den oberen Adreßleitungen anfangen kann, ist nur zu offensichtlich.

Es gibt bei der Peripherie-Adressierung allerdings eine Ausnahme, bei der auch das untere Adreßbyte ausgewertet wird: Die Adreßleitung A10 selektiert den Expansion-Port. Da man hier ziemlich viel anschließen kann, wird über das untere Byte zusätzlich die Auswahl des angesprochenen Gerätes vorgenommen. Das hätte aber genausogut umgekehrt sein können: Auswahl mit einer Leitung aus der unteren Hälfte und ausnahmsweise eine Zusatzauswahl mit dem oberen Adreßbyte.

Mit diesem Schönheitsfehler kann man noch am leichtesten leben. Block-I/O-Befehle werden ohnehin nur recht selten benötigt. Viel störender sind da schon das fehlende 8. Bit am Druckeranschluß oder die verwehrte externe RAM-Erweiterung.

Die PIO 8255

DIE ANSCHLUSSBELEGUNG DER PIO 8255



Dieser Peripheriebaustein wurde von INTEL bereits für den Vorgänger des Z80, die CPU 8080, entwickelt. Der 8255 ist aber so flexibel, daß er mit sehr vielen anderen Prozessoren zusammenarbeiten kann. Und er bietet ausreichend viele Möglichkeiten, um auch heute noch immer wieder eingesetzt zu werden.

Insgesamt verfügt der 8255 über 3 "Ports". Das sind Ein- und Ausgänge, mit denen Daten von und zu Peripheriebausteinen durchgeschaltet werden, die man nicht direkt an den Datenbus der CPU anschließen kann. Diese drei Tore sind dabei jeweils 8 Bit breit. Der Port C kann auch geteilt und zu Steuerzwecken für die beiden anderen Ports A und B benutzt werden. Von dieser Möglichkeit wird im CPC aber kein Gebrauch gemacht.

Unter den drei möglichen Betriebsarten hat man sich bei AMSTRAD für die einfachste und auch gebräuchlichste entschieden: Im Modus 0 werden alle drei Tore für Datenein- oder ausgaben benutzt. Dabei sind Port A und B jeweils nur als Ganzes in ihrer Datenrichtung einstellbar, Port C getrennt für seine obere und untere Hälfte.

Port A ist mit den Datenleitungen des Sound-Chips verbunden. Die Programmierung des Tongenerators erfolgt ausschließlich über die PIO, er hat keine eigene Adresse. Der Sound-Chip hat aber selbst auch noch eine Parallelschnittstelle implementiert, über die die Tastatur eingelesen wird.

Port B hat eine Vielzahl von Funktionen zu erfüllen. Allen ist aber gemein, daß hier nur Daten eingelesen werden.

Port C ist demgegenüber in beiden Hälften fest als Ausgang programmiert. Auch hier haben die einzelnen Datenleitungen wieder unabhängige Aufgaben.

Erklärung zu den Anschlußbezeichnungen:

Vcc und Vss

Über diese beiden Anschlüsse wird das IC mit Strom versorgt. Pin 7 (Vss) ist dabei der Masse-Anschluß. Als Versorgungsspannung begnügt sich die PIO wie alle anderen ICs auch mit +5 Volt.

CS – Chip Select

Nur wenn dieser Anschluß auf null Volt gelegt wird, sind die anderen Steuersignale RD, WR, A0 und A1 wirksam.

RESET

Wird an diesen Eingang ein positiver Spannungspegel angelegt, wird der Baustein initialisiert und in einen ungefährlichen Zustand gebracht. Das bedeutet vor allem, daß alle Ports auf Eingabe gestellt werden.

RD – Read

Ein Null-Pegel an diesem Eingang zeigt dem 8255 an, daß die CPU eins seiner Register lesen will. Der 8255 legt dann das durch A0 und A1 adressierte Register auf den Datenbus.

WR – Write

Mit Null-Legen dieses Pins werden die Register des 8255 beschrieben. Die auf dem Adreßbus liegenden Daten werden in das durch A0 und A1 adressierte Register der PIO eingelesen.

A0 und A1 – Adreßleitungen 0 und 1

Mit A0 und A1 wird bei einem Zugriff auf den 8255 zwischen Steuerregister und den drei Datenregistern unterschieden. Der 8255 verfügt über 4 interne Register. Die drei Datenregister sind 8 Bit breit und entsprechen direkt den Ports A, B und C. Das Steuerregister dagegen besteht nur aus 7 Bits. Dabei dient das unbenutzte 8. Bit als Signal, ob ein Schreibzugriff auf die Adresse des Steuerregisters auch wirklich dieses beschreiben soll. Alternativ dazu besteht nämlich noch die Möglichkeit, über diese Adresse einzelne Bits von Port C zu setzen oder zu löschen, was nichts anderes heißt, als daß die zugehörigen Pins auf 0 oder auf +5 Volt gelegt werden.

D0 bis D7 – Datenleitungen 0 bis 7

Diese 8 Leitungen werden an den Datenbus der CPU angeschlossen. Bei einem Zugriff auf den 8255 werden über diese Leitungen, entsprechend den Einstellungen an RD, WR, A0 und A1 die Daten in die jeweiligen Register geschrieben bzw. aus diesen gelesen.

PA0 bis PA7 – Port A, Leitungen 0 bis 7

Das sind die acht Anschlußleitungen des Ports A. Dieses Tor kann nur für alle acht Leitungen zusammen für Aus- oder Eingabe programmiert werden. Als Eingabeschnittstelle können die an den Leitungen anliegenden Spannungen jederzeit auf den Datenbus durchgeschaltet werden, wenn die CPU das entsprechende Register liest. Als Ausgang programmiert, wird ein Wert, den die CPU einmal in das zugehörige Register geschrieben hat, so lange auf diesen Leitungen als 0- oder +5-Volt-Pegel ausgegeben, bis die CPU den nächsten Wert in das Register schreibt. Ein als Ausgang programmierter Port kann auch gelesen werden. Dann erhält man den Wert, den die PIO gerade auf diesem Port ausgibt.

PB0 bis PB7 – Port B, Leitungen 0 bis 7

Völlig entsprechend die Leitungen für Port B.

PC0 bis PC7 – Port C, Leitungen 0 bis 7

Für Port C gilt wieder dasselbe wie für Port A. Zusätzlich kann Port C aber in zwei Hälften, für die Bits 0 bis 3 und 4 bis 7 getrennt für Ein- oder Ausgabe programmiert werden. Außerdem kann die CPU, durch bestimmte Schreib-

befehle ins Steuerregister, ganz geziehlt einzelne Bits dieses Tores löschen oder setzen. Bei den anderen Ports ist das nur über einen kleinen Umweg möglich. In den Betriebsmodi 1 und 2 bilden die Anschlüsse von Port C die Handshake-Signale für asynchronen Datentransfer.

DIE 3 VERSCHIEDENEN BETRIEBSARTEN DER PIO 8255

Für die verschiedenen Modi werden die Ports A, B und C auf zwei Gruppen aufgeteilt: Gruppe A besteht aus Port A und der oberen Hälfte von Port C, Gruppe B besteht aus Port B und der unteren Hälfte von C. Gruppe B kann nur in Modus 0 oder 1, Gruppe A in Modus 0, 1 oder 2 betrieben werden:

	Port A	Port B	Port C		Modus 0 1 2
Gruppe A	76543210	--------	7654	----	x x x
Gruppe B	--------	76543210	----	3210	x x –

Modus 0

Der auch im Schneider CPC verwendete Modus 0 ist der einfachste und üblichste. Alle Port-Leitungen der jeweiligen Gruppe stehen als Datenein- oder -ausgabeleitungen zur Verfügung. Die Ports A und B können aber nur als Ganzes für eine Datenrichtung eingestellt werden; Port C in zwei Hälften, da diese ja verschiedenen Gruppen angehören.

Modus 1

Modus 1 ermöglicht den Betrieb einer asynchronen, parallelen Schnittstelle, beispielsweise nach dem Centronics-Standard. Port A (bzw. B) wird als Ein- oder Ausgang definiert. Die entsprechende Hälfte von Port C dient nicht mehr dem Datentransfer, sondern stellt die verschiedenen Quittungssignale zur Verfügung (Strobe, Busy, Acknowledge o.ä.). Außerdem kann eine Datenleitung des Port C als Interrupt-Request-Signal für die CPU benutzt werden, wodurch ein Bedienen der Schnittstelle bei Bedarf möglich ist.

Modus 2

Diese Betriebsart ist nur bei Gruppe A möglich. Sie entspricht weitgehend Modus 1. Wieder stellt die entsprechende Hälfte von Port C die Steuersignale

zur Verfügung. Port A kann im Gegensatz zu Modus 1 jedoch gleichzeitig als Ein- und als Ausgang betrieben werden.

Die verschiedenen Modi werden über das Steuerregister eingestellt. Dazu muß Bit 7 des übermittelten Datenwortes immer gesetzt sein. Dann haben die Bits 0 bis 6 folgende Funktion.

Steuerregister: Bit 7 gesetzt:

Bit	Funktion	wenn 0	wenn 1
0	Port C, Bits 0 bis 3	Ausgang	Eingang
1	Port B	Ausgang	Eingang
2	Gruppe B	Modus 0	Modus 1
3	Port C, Bits 4 bis 7	Ausgang	Eingang
4	Port A	Ausgang	Eingang
5	Gruppe A	Modus 0	Modus 1
6	Gruppe A	Modus 0 oder 1	Modus 2

(Ist Bit 6 gesetzt, hat es Priorität über Bit 5.)

Wird das Steuerregister mit einem Datenbyte beschrieben, in dem Bit 7 nicht gesetzt ist, ist die Bit-Setz-Funktion für Port C angewählt. Diese Funktion ist in allen drei Modi für all die Bits aus Port C zulässig, die als Ausgang programmiert sind.

Steuerregister: Bit 7 gelöscht.

Bit	Funktion
Bit 0:	wird in das angewählte Bit von Port C kopiert: Bit 0 = 0 → löschen Bit 0 = 1 → setzen
Bits 1–3:	codieren binär die Nummer des Bits, das gesetzt oder gelöscht werden soll.
Bits 4–6:	ohne Funktion

ANSCHLUSS DER PIO 8255 AN DAS GESAMTSYSTEM

Die PIO ist im Schneider CPC noch spartanischer als der CRTC angeschlossen: Wäre nicht der Reset-Eingang, der bei einem positiven Signal aktiviert wird,

könnte man guten Gewissens behaupten, zum Ansteuern der PIO wäre kein einziges zusätzliches Gatter im CPC benötigt worden.

Das CS-Signal ist direkt mit A11 verbunden. Die Registerauswahl-Adressen A0 und A1 des 8255 sind mit A8 und A9 der CPU verbunden. Die Datenleitungen führen direkt auf den Datenbus.

Die Eingänge RD und WR der PIO liegen direkt an IORD und IOWR, zwei Signale, die im CPC für allgemeine Zwecke aus RD, WR und IORQ der CPU gewonnen werden. Bei einem I/O-Lesezugriff des Z80 werden RD und IORQ gleichzeitig low. Über ein Oder-Gatter verknüpft, ergeben sie IORD, das dann ebenfalls null wird. Das gleiche gilt für IOWR bei Schreibbefehlen.

Da die anderen Adressen im Schneider CPC ähnlich unvollständig durch eine einzige Adreßleitung codiert werden, müssen die restlichen Bits im oberen Adreßbyte alle gesetzt sein, damit sich nicht noch andere ICs angesprochen fühlen. Daraus ergeben sich folgende Adressen, mit denen die PIO angesprochen wird:

PIO:	A0	A1	CS		
CPU:	A8	A9	A11	Port-Adresse	Funktion
	0	0	0	&F4xx	Datenregister Port A
	0	1	0	&F5xx	Datenregister Port B
	1	0	0	&F6xx	Datenregister Port C
	1	1	0	&F7xx	Steuerregister
	x	x	1		nicht angesprochen

Port A – Output: &F4xx

Port A ist mit den Datenleitungen des Sound-ICs verbunden. Alle Programmdaten für den PSG müssen also über Port A der PIO gesendet werden. Port A ist im Schneider CPC standardmäßig als Ausgang programmiert.

Der PSG enhält jedoch seinerseits eine Parallelschnittstelle, über die die Tastatur eingelesen wird. 50mal in jeder Sekunde fragt das Betriebssystem die Tastatur ab.

Dann wird Port A jedesmal kurzzeitig auf Eingabe umgestellt, die Tastatur eingelesen und anschließend die Standardeinstellung für die Ports in das Steuerregister der PIO zurückgeschrieben.

Port B – Input: &F5xx

Dieses Tor ist im Schneider CPC für Eingaben eingestellt. Es mutwillig als Ausgang zu programmieren, kann unter Umständen Schaden anrichten. Die Funktion der einzelnen Bits (Leitungen) ist recht unterschiedlich:

Bit 0:

Der Anschluß PB0 ist mit dem Vsync-Signal des CRTC verbunden. Über diesen Anschluß kann man also jederzeit testen, wann der Kathodenstrahl im Monitorbild wieder von unten nach oben hochläuft.

Diese Zeit wird als *Frame Flyback* bezeichnet und ist ein besonders günstiger Zeitpunkt, um größere Veränderungen auf dem Bildschirm vorzunehmen. Viele Manipulationen an Bild und Bildausgabe würden zu unansehnlichen Flimmererscheinungen führen, wenn man sie mitten im Bild vornimmt. So wird das Farbblinken immer beim Strahlhochlauf vorgenommen, und auch die Routinen, die den Bildschirm scrollen, warten immer erst bis zum nächsten Frame Flyback.

Dazu muß man aber nicht immer das Programm unterbrechen. Die 300 Interrupts pro Sekunde im CPC werden mit dem Vsync-Signal synchronisiert. Zu jedem Strahlhochlauf wird garantiert ein Interrupt erzeugt. Die Software-Interrupts des Schneider-Betriebssystems bieten sogar die Möglichkeit, Ereignisse explizit nur für Frame-Flyback-Interrupts zu programmieren. Die selbstprogrammierte Sprite-Bewegungsroutine sollte also nicht selbst auf den nächsten Vsync-Impuls warten und die CPU unausgenutzt im Kreise laufen lassen, sondern einen Interrupt programmieren, der beim nächsten Strahlhochlauf das Sprite bewegt.

Bits 1, 2 und 3:

Diese Leitungen sind an drei Drahtbrücken auf der Platine des Schneider CPC angeschlossen, die je nachdem, ob es nun wirklich ein Schneider oder ein AMSTRAD oder sonst eine Firma ist, anders gesetzt sind. Tatsächlich gibt es nur einen gravierenden Unterschied zwischen einem Original-AMSTRAD- und einem Schneider-Computer: Diese Drahtbrücken sind anders gesetzt. Nur danach entscheidet sich, welche Marke in der Einschaltmeldung genannt wird. Dabei entspricht:

Brücke = 0
keine Brücke = 1

Und folgende Firmen stehen zur Auswahl:

Bit 1	2	3	Firma
0	0	0	Isp
0	0	1	Triumph
0	1	0	Saisho
0	1	1	Solavox
1	0	0	Awa
1	0	1	Schneider
1	1	0	Orion
1	1	1	Amstrad

Die Brücken befinden sich direkt neben der PIO auf der Platine und sind mit LK1 bis LK4 für Bit 1 bis Bit 4 gekennzeichnet. Für die Einschaltmeldung sind aber nur LK1 bis LK3 maßgebend.

Außer der individuellen Firmenmeldung läßt sich mit diesen Bits natürlich noch mehr anstellen. Da Firmen wie Schneider nur als nationale Unterhändler im jeweiligen Verkaufsland auftreten, kann man aus der Firma auf die Nationalität des Käufers schließen. Ein Programm könnte beispielsweise diese Brücken testen und daraufhin automatisch die Dialogtexte in der entsprechenden Landessprache verfassen.

Bit 4:

Dies ist die vierte Brücke, die je nach Verkaufsland gesetzt sein kann oder auch nicht. Auch dieses Bit wird nur beim Einschalten des Computers ausgewertet: Fehlt sie (Bit 4 = 1) wird der CRTC für PAL oder SECAM initialisiert. Ist sie gesetzt (Bit 4 = 0), erzeugt der CRTC ein NTSC-kompatibles Signal, wodurch der "Schneider CPC" auch in den USA verkaufbar ist.

Solange der Computer nur mit dem mitgelieferten Monitor betrieben wird, ist das eigentlich egal. Interessant wird es aber, wenn man ihn mittels Modulator an den eigenen (Farb-) Fernseher anschließen will. Dann muß das Timing schon stimmen. Dabei gibt es bei der Programmierung des CRTC zwischen PAL und SECAM keinen Unterschied. Deren Schwarzweißnorm ist ja auch identisch.

Die verschiedene Farbmodulation findet im Modulator-Netzteil statt. Ein in Frankreich verkaufter Modulator dürfte also kaum für einen deutschen Fernseher zu gebrauchen sein, auch wenn der CRTC in beiden Fällen die gleichen Signale liefert.

Bit 5:

Diese Leitung wird vom Betriebssystem nie abgefragt. Sie führt zu einem Anschluß am Expansion-Port an der Rückseite des Computers, der EXP benannt ist, und wird im CPC 464 ansonsten nicht benutzt.

Anders jedoch im CPC 664 und 6128 und, sobald man am CPC 464 einen AMSDOS-Controller anschließt: Hier ist diese Leitung an eine weitere Drahtbrücke angeschlossen. Normalerweise ist diese Brücke nicht eingesetzt. Dann hat das AMSDOS-ROM die Nummer 7 und verhält sich wie ein normales Hintergrund-ROM.

Setzt man diese Brücke jedoch ein, verändert sich die ROM-Select-Adresse des AMSDOS-Controllers: Sie wird 0 und unterdrückt damit das eingebaute BASIC-ROM. Schaltet man den Computer ein, wird AMSDOS wie ein Vordergrund-ROM behandelt und anstelle von BASIC gestartet. Das macht natürlich nur einen Sinn, wenn AMSDOS diese Änderung erkennt. Dafür könnte es Bit 5 von Port B testen. Das tut es aber nicht, sondern fragt das Betriebssystem direkt, welche Nummer es hat. Ist sie 0, wird automatisch CP/M gebootet. Somit wurde dieser Anschluß wohl unnütz vertan.

Bit 6:

Dieser Pin ist mit der Busy-Leitung des Druckeranschlusses verbunden. Bevor man ein Zeichen zum Drucker schicken kann, muß man erst nachfragen, ob dieser überhaupt bereit ist, ein Zeichen zu empfangen. Während er druckt, ist das beispielsweise meist nicht der Fall. Würde man das Busy-(Beschäftigt-) Signal des Druckers ignorieren, würden ziemlich viele Zeichen beim Ausdruck verlorengehen. Zeichen dürfen erst gesandt werden, wenn der Drucker "ready" ist, das heißt, wenn diese Leitung auf Null-Pegel liegt.

Das Betriebssystem testet diese Leitung natürlich automatisch, bevor es ein Zeichen absetzt. Durch Testen von Bit 6 in Port B kann man jedoch auch in BASIC schon vorher feststellen, ob ein Drucker angeschlossen und bereit ist, bevor man einen Text abschickt:

```
IF (INP(&F5FF)and 128)=0 THEN PRINT#8,.....
```

Bit 7:

Über die letzte Leitung von Port B wird das verstärkte Signal vom Kassettenrecorder eingelesen. Da die Zeiten immer knapp sind, wurde hierfür Bit 7 gewählt. Das läßt sich neben Bit 0 am schnellsten testen: Einmal nach links rotiert, und schon ist es im Carry-Flag. Daß die Elektronik hierbei das analoge

Eingangssignal vom Wiedergabekopf des Recorders in negative Halbwelle (0) und positive Halbwelle (1) trennt, schadet nicht und ist sogar erwünscht: Alle Aufzeichnungsverfahren auf magnetisierbaren Datenträgern, auch auf Disketten, codieren die Bits der zu speichernden Dateien mit verschieden langen 0- oder 1-Halbwellen. Die Halbwellen haben dabei, da sie digital erzeugt sind (nämlich aus einem 0-1-Signal), idealerweise Rechteckform.

Port C – Output: &F6xx

Im Gegensatz zu Port B ist dieses Tor immer für Ausgaben programmiert. Es für Eingaben umzuprogrammieren bringt nichts, ist aber vollkommen ungefährlich.

Bits 0 bis 3:

Die Tastatur wird vom Betriebssystem via Interrupt 50mal in der Sekunde abgefragt. Ihr Status wird dabei über den PSG und Port A der PIO eingelesen. Damit sind aber maximal 8 verschiedene Tasten zu erfassen, die den 8 Bits des eingelesenen Datenbytes entsprechen. Die Nummer "einer gedrückten Taste" direkt einzulesen, ist nicht möglich, da mehrere gleichzeitig gedrückt werden können.

Deswegen ist die gesamte Tastatur in einer Matrix organisiert: 8 Spalten breit und 10 Zeilen hoch. Über die Bits 0 bis 3 muß vor jedem Lesen der Tastatur die gewünschte Zeile angesprochen werden. Eine vollständige Tastaturabfrage liefert also erst einmal nur 10 verschiedene Bytes, in denen eventuell das ein oder andere Bit gesetzt ist.

Das Aktivieren eines Zeilendrahtes, entsprechend der 4-Bit-Nummer, übernimmt dabei ein BCD-Decoder-IC. Nur Nummern im Bereich 0 bis 9 werden akzeptiert. Ist die hier ausgegebene Zeile größer als 9, wird überhaupt kein Zeilendraht aktiviert.

Dies ist insofern interessant, weil dadurch der Port des PSG, über den die Spaltendrähte der Tastatur eingelesen werden, gefahrlos als Ausgang programmiert werden kann, um für irgendwelche Spielereien beispielsweise aus dem Joystick-Eingang einen Ausgang zu machen. Man muß für diese Zeit nur den Interrupt abstellen, da eine Tastaturabfrage zwischendurch wieder Zeilendrähte aktiviert.

Bit 4:

Über diesen Ausgang wird der Motor des Datenrecorders ein- und ausgeschaltet. Bei den CPCs 664 und 6128 wird hiermit der Remote-Ausgang ge-

steuert. Eine Null an diesem Bit stoppt den Rekorder, eine Eins startet den Motor, vorausgesetzt, daß auch die PLAY-Taste gedrückt ist.

Bit 5:

Diese Leitung ist das Pendant zu Bit 7 von Port B. Hierüber wird das Rechtecksignal ausgegeben, das dann auf der Datenkassette gespeichert werden soll. Da beim Speichern der Daten leichter die erforderlichen Zeiten eingehalten werden können als beim Laden, muß der Ausgang nicht unbedingt auf einem günstigen Bit 0 oder 7 liegen. Schöner wäre es natürlich schon gewesen.

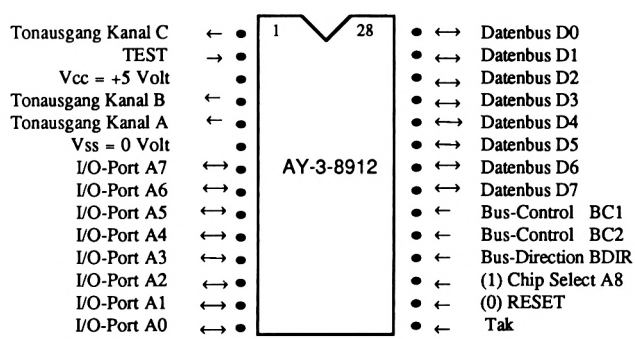
Bit 6 und 7:

Der Datenbus des PSG ist an Port A der PIO angeschlossen. Um ihn zu programmieren, müssen aber auch seine Eingänge BC1 und BDIR entsprechend gesetzt werden:

PIO: PSG:	Bit 6 BC1	Bit 7 BDIR	Funktion:
	0	0	Datenwort wird ignoriert
	0	1	in adressiertes Register schreiben
	1	0	aus adressiertem Register lesen
	1	1	Register adressieren

Der PSG: AY-3-8912

DIE ANSCHLUSSBELEGUNG DES AY-3-8912



PSG steht für "Programmable Sound Generator". Es handelt sich hierbei also um das IC, das im Schneider CPC der Tonerzeugung dient. Dieses IC von General Instruments ist dabei besonders vielseitig einsetzbar, weil es sehr viele Funktionen enthält, die alle durch die CPU programmierbar sind.

Außerdem ist in diesem PSG noch eine bidirektionale Parallelschnittstelle implementiert, also ein Tor zur Außenwelt, durch das Daten sowohl ausgegeben als auch gelesen werden können. Dieser Port wird im Schneider aber ausschließlich für Eingabezwecke benutzt: An ihm ist die Leseseite der Tastenmatrix angeschlossen. Diesen Port also als Ausgang zu programmieren, ist beim CPC nur in Ausnahmefällen sinnvoll.

Der AY-3-8912 ist sehr einfach in ein System zu integrieren: Er benötigt nur eine Spannungsversorgung von 5 Volt und einen Eingangstakt. Was er machen soll, bekommt er über 8 Datenleitungen und zwei Steuerleitungen mitgeteilt, die allesamt an die PIO angeschlossen sind.

Dabei werden alle Signale im PSG durch Teilen eines Eingangstaktes erzeugt, der im Schneider CPC genau 1 MHz beträgt.

Die Möglichkeiten dieses Sound-Chips sind sehr umfassend: Er hat drei Tonkanäle, die getrennt programmiert werden können. Im Schneider CPC werden sie in zwei Gruppen aufgeteilt und dem Stereo-Ausgang zugeführt: A+B/2 bilden den linken Kanal, B/2+C den rechten. Für den eingebauten Lautsprecher werden aber alle drei Kanäle zusammengefaßt.

Für jeden Tonkanal kann getrennt die Lautstärke und Frequenz eingestellt werden; die Lautstärke dabei in 15 Stufen, die sogar ein logarithmisches Raster haben, was dem menschlichen Hörempfinden sehr entgegenkommt. Die Frequenz wird allerdings in einer linearen Skalierung angegeben, obwohl hier das logarithmische Raster noch viel mehr angebracht wäre. Der Grund für die lineare Teilung der Tonperioden-Längen ist, daß alle Töne durch Teilen des Eingangstaktes erzeugt werden.

Für alle Kanäle zusammen gibt es auch noch einen Hüllkurvengenerator. Wählt man diesen an, wird die Lautstärke des entsprechenden Kanals von ihm bestimmt. Man hat dabei die Auswahl zwischen acht verschiedenen Hüllkurvenformen, die alle Kombinationen aus fallenden und steigenden Sägezahnflanken sind. Dabei kann man die Geschwindigkeit, mit der die Hüllkurven abgearbeitet werden sollen, in weiten Grenzen einstellen.

Ebenfalls für alle Kanäle gemeinsam ist ein Rauschgenerator. Dessen Grundfrequenz ist in 32 Schritten einstellbar. Mit ihm kann man beispielsweise sehr realistische Knall- und Schußgeräusche oder die Percussion zu einem Musikstück realisieren.

Erklärung zu den Anschlüssen:

Vcc und Vss

Die Stromversorgung des PSG erfolgt über die mit Vcc und Vss benannten Eingänge. Dabei wird an Pin 3 (Vcc) +5 Volt und an Pin 6 (Vss) 0 Volt, also Masse angeschlossen.

TEST

Der Eingang TEST wird im Betrieb nicht beschaltet. Er dient dazu, den PSG nach der Herstellung auf Funktionstüchtigkeit zu prüfen.

A, B, C

An den Pins 1, 4 und 5 liegt das Ausgangssignal der Tonkanäle C, B und A an. Diese werden im Schneider CPC über Widerstände für den Stereo-Ausgang und den eingebauten Lautsprecher teilweise wieder zusammengemischt.

A0 bis A7

Die Anschlüsse 7 bis 14 sind der im PSG integrierte bidirektionale I/O-Port. Der Port kann aber nur als Ganzes für Ein- oder Ausgabe programmiert werden. Im Schneider CPC wird er benutzt, um die Tastatur einzulesen. Die Bezeichnungen A0 bis A7 könnten irrigerweise auf einen Adreßanschluß hindeuten. Es ist aber der (Daten-) Port des AY-3-8912. Das A resultiert daher, daß ein verwandtes IC zwei Ports hat, die A und B genannt wurden.

D0 bis D7

Pins 21 bis 28 sind die Anschlüsse an den Datenbus und damit auch zur CPU. Hierüber wird der PSG programmiert und Daten von und zum Port übermittelt. Im Schneider CPC sind aber auch diese Anschlüsse über die PIO geführt.

Takt

Pin 15 ist der Takteingang, dessen Frequenz beim Schneider CPC 1 MHz beträgt.

Reset

Wird Pin 16 auf Masse gelegt, so wird der PSG neu initialisiert und die Tonerzeugung auf allen Kanälen abgestellt.

A8

Um den PSG anzusprechen, muß an Pin 17, Chip Select, eine positive Spannung anliegen. Andernfalls haben die Steuersignale BC1, BC2 und BDIR keine Wirkung. Dieser Eingang ist beim Schneider CPC fest auf +5 Volt gelegt.

BDIR

Über den Eingang BDIR (Bus Direction) wird festgelegt, ob ein Lesezugriff oder ein Schreibzugriff auf die internen Register erfolgen soll. Liegt an BDIR ein positives Signal an, werden Daten zum PSG gesandt.

BC1 und BC2

Die beiden Eingänge BC1 und BC2 (Bus Control) steuern die Verwendung des auf dem Adreßbus liegenden Datenwortes.

FUNKTIONSAUSWAHL IM PSG ÜBER BC1, BC2 UND BDIR

(A8 = Chip Select = 1)

BC1	BC2	BDIR	Funktion:
0	0	0	Datenwort wird ignoriert
0	0	1	Register adressieren
0	1	0	Datenwort wird ignoriert
0	1	1	in adressiertes Register schreiben
1	0	0	Register adressieren
1	0	1	Datenwort wird ignoriert
1	1	0	aus adressiertem Register lesen
1	1	1	Register adressieren

Wie man sieht, sind viele Funktionen doppelt vorhanden. Andererseits erzeugen viele Kombinationen keinen Schreib- oder Lesevorgang. Das erklärt sich

daraus, daß der AY zuerst für die CPU 1610 von General Instruments entworfen wurde und man deshalb bei der Gestaltung der Anschlüsse auf deren spezielle Bedürfnisse Rücksicht genommen hat.

Nimmt man die Tabelle noch etwas genauer unter die Lupe, so erkennt man, daß alle Funktionen mit nur zwei Steuerleitungen ausgewählt werden können, nämlich mit BC1 und BDIR, wenn man BC2 auf +5 Volt legt:

BC1	BC2	BDIR	Funktion:
0	0	0	Datenwort wird ignoriert
0	1	1	in adressiertes Register schreiben
1	1	0	aus adressiertem Register lesen
1	1	1	Register adressieren

Das wird deshalb auch fast immer gemacht. Auch im CPC liegt der Anschluß BC2 direkt auf +5 Volt. Nur BC1 und BDIR werden benutzt.

Im CPC ist der PSG an die PIO angeschlossen: Die Leitungen zum Datenbus sind nicht direkt auf den Datenbus der CPU gelegt, sondern an Port A der PIO. Die Steuerleitungen werden über Kanal C angesprochen: BDIR mit Bit 7 und BC1 mit Bit 6. Der PSG hat deshalb im Schneider CPC keine eigene Port-Adresse. Um ihn zu programmieren, muß man sich immer an die PIO wenden.

Portadresse PIO			PSG
&F4xx	Port A (I/O)	◄——►	Datenbus
&F6xx	Port C (-/O) Bit 7	►	BDIR
	Bit 6	►	BC1

Wie bereits erwähnt, läßt sich der AY-3-8912 programmieren. Dazu müssen die Register im PSG beschrieben werden. Wie man an den Funktionen der Steuerleitungen erkennt, muß man dazu erst ein Register adressieren, bevor man es in einem zweiten Schreibzyklus beschreiben kann.

Da sowohl der Datenbus als auch die Steuereingänge des PSG an die PIO angeschlossen sind, ergibt sich, daß es ein recht kompliziertes Unterfangen ist, ein Register des PSG korrekt zu beschreiben oder einen Zeilendraht der Tastaturmatrix einzulesen.

Im Schneider CPC hat man deshalb extra einen Vektor eingerichtet, mit dem man ein Register des PSG einfach programmieren kann: MC Sound Register. Diese Routine liegt beim CPC 464 im unteren ROM ab Adresse &0826:

Befehl		Anmerkung
		Einsprung mit A = Registernummer und C = zu programmierender Wert
DI		Interrupt verbieten
		1. PSG-Register adressieren
LD	B,#F4	Adresse von Port A der PIO
OUT	(C),A	Register A (PSG-Reg.-Nr.) in Port A der PIO schreiben
LD	B,#F6	Adresse von Port C der PIO. Hier Bit 7=BDIR und Bit 6=BC1
IN	A,(C)	Port C Zustand lesen
OR	#C0	Bit 6 und 7 setzen → BDIR=1 und BC1=1 → PSG-Register adressieren
OUT	(C),A	in Port C beschreiben (Adreß-Strobe für den PSG) (der PSG latcht jetzt die Registeradresse von Port A ein).
AND	#3F	Bit 6 und 7 zurücksetzen → BDIR=0 und BC1=0 → inaktiv
OUT	(C),A	in Port C schreiben
		2. PSG-Register beschreiben
LD	B,#F4	Adresse von Port A der PIO
OUT	(C),C	Z80-Register C (PSG-Reg.-Wert) in Port A der PIO s schreiben
LD	B,#F6	Adresse von Port C der PIO → Bit 6 und 7=BC1 und BDIR
LD	C,A	C = alter Wert aus Port C mit BDIR und BC1=0 (inaktiv)
OR	#80	A = Bit 7 setzen → BDIR=1 und BC1=0 → PSG-Register beschreiben
OUT	(C),A	A nach Port C schreiben (Daten-Strobe für den PSG) (der PSG latcht jetzt den Wert aus Port A ins angewählte Register)
OUT	(C),C	C nach Port C schreiben → BDIR und BC1=0 → inaktiv
EI		Interrupt wieder zulassen
RET		zurück zum rufenden Programm

Zum weiteren Verständnis dieser Routine sei noch folgendes angemerkt: Die Port-Adresse der PIO ist binär %111101xx????????. Das elfte Bit, die 0, adressiert die PIO, die Bits 8 und 9 und xx wählen, je nach Wert, Port A, B, C oder das Steuerregister an. Das niederwertige Byte ???????? wird nicht ausgewertet! Beim Befehl OUT (C),A z.B. wird das Register C auf der unteren Adreßhälfte ausgegeben, die aber nicht interessiert! Benutzt wird der "Nebeneffekt", daß dabei Register B auf der oberen Adreßhälfte ausgegeben wird. Obwohl im Befehls-Mnenonic also das C-Register angegeben ist, wird eigentlich nur mit B adressiert.

DIE REGISTER DES AY-3-8912

Insgesamt hat der PSG 16 verschiedene Register, wovon aber eins der implementierte I/O-Port und ein weiteres ein nicht existenter zweiter Port ist. Den zweiten Port gibt es nur in einer anderen Ausführung dieses ICs, dem AY-3-8910.

Register	Belegung	Funktion
0	xxxxxxxx	LSB der Tonperiodenlänge
1	 xxxx	MSB für Kanal A
2	xxxxxxxx	LSB der Tonperiodenlänge
3	 xxxx	MSB für Kanal B
4	xxxxxxxx	LSB der Tonperiodenlänge
5	 xxxx	MSB für Kanal C
6	... xxxxx	Rauschperiodenlänge
7	. xxxxxxx	Kontrollregister
8	... hxxxx	Lautstärke Kanal A
9	... hxxxx	Lautstärke Kanal B
10	... hxxxx	Lautstärke Kanal C
11	xxxxxxxx	LSB der Periodenlänge des
12	xxxxxxxx	MSB Hüllkurvengenerators
13	 xxxx	Hüllkurvenform
14	xxxxxxxx	I/O-Port

Die Tonperioden-Register (Register 0 bis 5)

Der Eingangstakt (CPC: 1MHz) des PSG wird in einer ersten Stufe vorab durch 16 geteilt. Danach wird er noch einmal für jeden Kanal durch eine programmierbare Teilerkette auf die gewünschte Periodenlänge geteilt und so dem jeweiligen Kanal zugeführt.

Das Teiler-Verhältnis kann dabei in $2^{12} = 4096$ Stufen eingestellt werden. Da der Datenbus nur acht Bit breit ist, mußte die Programmierung der Tonperiode auf jeweils zwei Register pro Kanal verteilt werden. Das erste Register enthält dabei die unteren acht Bits, das zweite Register jeweils die oberen vier Bits.

Die Frequenz eines Kanals ergibt sich dabei zu:

$$f = \frac{f(takt)}{16 * tv}$$

wobei f(takt) die Frequenz des Eingangstaktes ist, die durch 16 geteilt wird, und tv das programmierte Teilerverhältnis.

Für den internationalen Kammerton A mit 440 Hertz ergibt sich deshalb etwa:

$$f(A) = 440 = \frac{1.000.000}{16 * tv} \Leftrightarrow tv = \frac{1.000.000}{16 * 440} = 142{,}045$$

Zur Programmierung des PSG müssen die Nachkommastellen natürlich weggerundet werden, wodurch sich, vor allem in der obersten Oktave, hörbare Dissonanzen ergeben können.

Die erreichbare niedrigste Frequenz ist:

$$f(min) = \frac{1.000.000}{16 * 4095} = 16{,}28 \text{ Hertz}$$

Die erreichbare höchste Frequenz liegt bei 62500 Hertz. Sie ist aber nicht so interessant, weil der nächsttiefere Ton bereits eine ganze Oktave entfernt liegt. Ein musikalisch verwertbares Raster der Frequenzen ergibt sich erst unter 4000 Hertz.

Eine Besonderheit stellt noch die Tonperiodenlänge 0 dar: Bei diesem Teilerverhältnis wird kein Rechtecksignal mehr erzeugt, sondern der Ausgang ständig auf dem oberen dem der Amplitude entsprechenden Ausgangspotential gehalten. Dadurch kann man dann durch Programmierung der Amplitude des Kanals beliebige Signalformen erzeugen.

Alle Tonsignale des PSG werden durch Teilen des Eingangstaktes und, davon abhängig, Umschalten des Ton-Ausganges zwischen 0 und der eingestellten

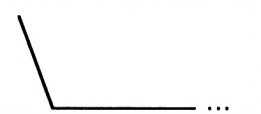
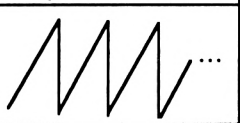
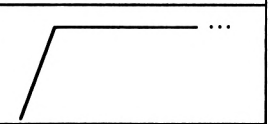
Amplitude erzeugt. Der PSG erzeugt deshalb ausschließlich Rechteck-Signale. Diese sind, im Vergleich zu den meisten natürlichen Instrumenten, sehr oberwellenreich. Deshalb muß man hier an Harmonien sehr viel höhere Anforderungen stellen. "Gewagte" Dreiklänge, die auf einem Klavier durchaus harmonisch klingen, wirken beim AY-3-8912 meist äußerst disharmonisch und "schräg".

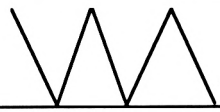
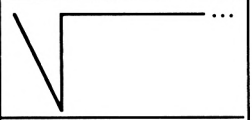
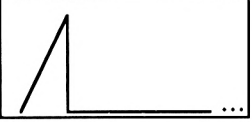
Das Kontrollregister (Register 7)

Bit	0	1	Bedeutung
0	ja	nein	Tonausgabe auf Kanal A
1	ja	nein	Tonausgabe auf Kanal B
2	ja	nein	Tonausgabe auf Kanal C
3	ja	nein	Rauschen auf Kanal A zumischen
4	ja	nein	Rauschen auf Kanal B zumischen
5	ja	nein	Rauschen auf Kanal C zumischen
6	in	out	Richtung des I/O-Ports

Die möglichen Hüllkurvenformen (Register 13)

In der folgenden Tabelle sind die möglichen Hüllkurvenformen des AY-3-8912 dargestellt und die Zahl, mit der Register 13 dafür jeweils programmiert werden muß, in binärer Form. Bei zwei Hüllkurven gibt es mehrere Zahlen, durch die sie erzeugt werden können. Alle Hüllkurven lassen sich aber auch nur mit den letzten drei Bits 0, 1 und 2 erzeugen, wenn man Bit 3 immer auf 1 setzt.

Nummer	Form	Nummer	Form
1000		1001 00xx	
1100		1101	

Nummer	Form	Nummer	Form
1010		1011	
1110		1111 01xx	

Hüllkurven-Perioden-Register (Register 11 und 12)

Der Verlauf des Amplitudenanstiegs und -abfalls wird natürlich auch vom Eingangstakt gesteuert. Hierbei wird ebenfalls der bereits durch 16 geteilte Takt verwendet. Dieser wird dann über die 16 Bit breite programmierbare Teilerkette auf die gewünschte Geschwindigkeit herabgesetzt.

Mit jeder Periode dieses Signals wird dann die Amplitude verändert. Da insgesamt 16 verschiedene Amplituden möglich sind, ist ein einzelner Sägezahn nach 16*16*tv=256*tv Takten abgearbeitet. tv ist hierbei wieder das programmierte Teilerverhältnis.

Die kürzeste programmierbare Sägezahn-Periode hat eine Frequenz von

$$f(max) = \frac{1.000.000}{256 * tv} = \frac{1.000.000}{256 * 1} = 3906\ Hz$$

Werden so kurze Sägezahn-Perioden programmiert, daß sie im hörbaren Bereich liegen, so nimmt dies unser Ohr nicht mehr als ein Vibrato, sondern als ein eigenständiges Frequenzsignal wahr. Der abgestrahlte Ton scheint deshalb ein Mischsignal aus der eigentlichen Tonperiodenlänge und der Länge der Sägezahn-Periode zu sein.

Programmiert man die Tonperiode eines Kanals, dessen Amplitude über den Hüllkurven-Generator gesteuert wird, mit dem Wert 0, so wird die Ausgangsspannung dieses Kanals nur noch durch den Amplitudenverlauf bestimmt. Damit kann man dann (mit gewissen Abstrichen) Sägezahn- oder Dreieck-Signale erzeugen.

Sinnvoll sind aber meist nur Periodenlängen, die weit unter der Hörgrenze liegen. Die längste erzeugbare Periodenlänge ist:

$$\frac{1}{p(max)} = f(min) = \frac{1.000.000}{256 * 65535} \quad <=> \quad p(max) = \frac{256 * 65535}{1.000.000} = 16.777 \text{ Sek.}$$

Die Lautstärken-Register (Register 8, 9 und 10)

Beim AY-3-8912 kann jeder Kanal mit einer konstanten Amplitude (Lautstärke) in 15 Stufen programmiert werden. Programmiert man ein Lautstärken-Register mit dem Wert 0, wird dieser Kanal abgeschaltet.

Dabei ist das Lautstärkenraster logarithmisch angelegt, wodurch beim menschlichen Gehör der Eindruck eines gleichmäßigen Lautstärke-Anstiegs entsteht. Wird in einem Lautstärken-Register aber Bit 4 gesetzt, das in der Tabelle mit h gekennzeichnet ist, so wird der Hüllkurvengenerator zur Erzeugung der Lautstärke angewählt.

Der Lautstärke-Unterschied zwischen zwei aufeinanderfolgenden Amplitudenwerten beträgt beim AY-3-8912 etwa 2.85 dB. Die folgenden Pegel wurden mit Hilfe eines Kassettenrecorders und dessen Aussteuerungsanzeige ermittelt. Absolute Genauigkeit sollte deshalb nicht erwartet werden.

Man sieht aber, daß die in Dezibel angegebenen Signalamplituden einen halbwegs konstanten Abstand zueinander haben. Da die Bezeichnung Dezibel eine logarithmische Skalierung in eine lineare übersetzt (Verdoppelung = 10 dB), muß die Amplitudenabstufung also tatsächlich logarithmisch sein:

Amplitude	Pegel
15	5.0
14	2.5
13	0.0
12	–3.0
11	–5.5
10	–8.5
9	–13.0
8	–15.0
7	–18.0
8	–21.0

Der Video-Controller HD 6845

Auch der HD 6845 von Motorola reiht sich wieder unauffällig zwischen die anderen ICs ein. Obwohl er sehr leistungsfähig ist, ist er bei seinen Ansprüchen an die Einbindung in das Gesamtsystem recht genügsam: Wie alle anderen ICs auch, begnügt er sich mit einer einfachen Spannungsversorgung von +5 Volt und einem einfachen Takteingang. Für die Auswahl seiner 18 Register beansprucht er nur 2 Adressen.

Zusammen mit dem Gate Array übernimmt er die Hauptarbeit bei der Erzeugung der Monitorsignale. Die oft für einen Video-Controller synonym gebrauchte Bezeichnung CRTC ist eine Abkürzung für "Cathode Ray Tube Controller". Damit kommt zum Ausdruck, daß dieses IC wichtige Signale für die Bilddarstellung auf Kathodenstrahl-Röhren, also Monitor oder Fernseher, liefert. Der CRTC ist das Bindeglied zwischen Bildwiederholspeicher (RAM) und Monitor.

Dabei sorgt sich der HD 6845 weniger um die farbliche Ausgestaltung des Bildschirms und auch nicht darum, daß die einzelnen Bits aus den Bildschirm-Bytes zur richtigen Zeit zum Monitor gelangen. Diese Aufgabe wird vom Gate Array übernommen.

Aufgabe des CRTC ist es, die Rahmenbedingungen für das Monitorbild zu schaffen: Er erzeugt die Signale für die vertikale und horizontale Synchronisation des Bildes, zeigt an, wann beschreibbare Bildschirmteile dargestellt werden und adressiert dann auch den Bildwiederholspeicher.

Außerdem hat er noch einige weitere Funktionen, die im CPC aber nicht benutzt werden: Hardware-mäßiger Cursor und Lightpen-Eingang.

Das Cursor-Signal wird an einem separaten Pin ausgegeben und ist sogar auf den Expansion-Port durchgeführt. Es dürfte aber schwerfallen, dafür eine sinnvolle Verwendung zu finden (vielleicht in Verbindung mit einem Lightpen). Normalerweise benutzt man es, um entweder im Character-ROM einen zweiten Zeichensatz anzuwählen (etwas dickere oder invertierte Zeichen) oder direkt dem darzustellenden Zeichen zu überlagern (Invertieren, Misch-Addition o.ä.). Auf jeden Fall wird immer dann, wenn das Cursor-Signal aktiv wird, irgend etwas an der Zeichenausgabe verändert, um die Cursor- Position auf dem Bildschirm sichtbar zu machen.

Beim Schneider CPC ist ein Hardware-Cursor ohne Aufwand nur im Bildschirmmodus 1 realisierbar, da das Cursorsignal immer nur für ein Zeichen aktiv wird. Der CRTC setzt aber immer zwei Bytes einer Buchstabenposition

gleich. Die Gründe dafür werden nachher noch ausführlicher behandelt werden. So würde in Modus 0 nur die linke Hälfte der Zeichenposition markiert, in Modus 2 dafür aber 2 Zeichen gleichzeitig. Das liegt daran, daß in den unterschiedlichen Bildschirmmodi 1, 2 bzw. 4 nebeneinanderliegende Bytes im Bildschirmspeicher für einen Buchstaben zuständig sind.

Der Lightpen-Eingang ist ebenfalls nur zum Systembus-Anschluß durchgeführt. Die Einsatzmöglichkeiten dieses Eingangs sind stark beschränkt, weil der HD 6845 nur Informationen über die Buchstabenposition des Lightpens liefert. Menü-Auswahl ist aber ohne weiteres mit einem 5-Mark-Selbstbau-Lightpen möglich.

Mit Hilfe des HD 6845 ist es möglich, einen Computer in Ländern mit verschiedener Fernsehnorm zu verkaufen: Alle Synchronisationssignale sind in weiten Grenzen programmierbar. So ist der Schneider CPC darauf vorbereitet, sowohl PAL- und SECAM-kompatible (Deutschland) als auch NTSC-kompatible (Amerika) Signale zu erzeugen. Für den Monitor wäre es eigentlich egal. Man kann den CPC aber auch mittels Modulator oder Scart-Buchse an einen Fernseher anschließen.

Je nachdem, in welchem Land ein AMSTRAD-Computer verkauft wird, ist eine einzige Drahtbrücke auf der Platine anders gesetzt. Diese Brücke ist mit Bit 4 von Port B der PIO verbunden. Beim Initialisieren des Rechners wird diese Leitung abgefragt und abhängig davon, ob die Brücke gesetzt ist oder nicht, stellt sich der Rechner auf PAL oder SECAM ein.

Im CPC ist der CRTC auf eine recht unübliche Art eingesetzt: Normalerweise ist er nämlich mehr für Textsysteme gedacht. Dabei steht im Bildschirmspeicher nur der Code des darzustellenden Zeichens.

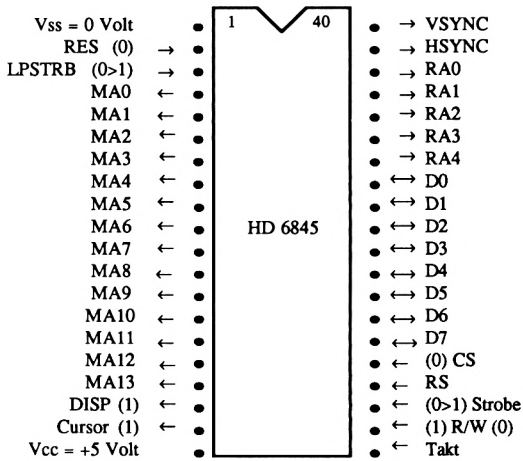
Mit dessen Hilfe und der aktuellen Rasterzeile innerhalb des Buchstabens wird ein Zeichen-ROM adressiert, aus dem dann erst das Bitmuster für den Monitor ausgelesen wird. Außerdem kann auch noch das Cursor-Signal, wie oben aufgeführt, zur Adressierung des Character-ROMs benutzt werden.

Mit seinen 14 Adreßleitungen kann der HD 6865 einen Textspeicher von bis zu 16000 Buchstaben verwalten, weit mehr, als auf einen normalen Monitor passen. Weil man aber auch den Anfang in diesem Textspeicher programmieren kann, kann man das Monitorbild recht einfach durch einen größeren Text hindurchscrollen lassen, ohne im Textspeicher selbst etwas ändern zu müssen.

Beim Schneider CPC ist der Bildschirmspeicher aber für 100%ige Grafikdarstellung vorgesehen. Zu diesem Zweck wird der CRTC in einer recht unüb-

lichen Art beschaltet: Die Adreßleitungen, mit denen der CRTC eine Rasterzeile im Zeichen-ROM adressieren will, werden einfach selbst zum Adressieren des RAMs benutzt.

ANSCHLUSSBELEGUNG DES CRTC HD 6845:



Erklärung zu den Anschlüssen:

Vcc und Vss

Über diese beiden Anschlüsse wird der CRTC mit Strom versorgt. Pin 1 (Vss) ist dabei der Masseanschluß.

Takt

Wie beim PSG werden durch programmierbare Teilerketten alle Ausgangssignale aus einem einzigen Eingangstaktsignal erzeugt.

Die Taktfrequenz bestimmt daher mit, welche Dauer die vom CRTC erzeugten Signale haben.

RES – Reset

Ein Null-Signal an diesem Pin versetzt den CRTC in einen definierten Ausgangszustand. Das Signal ist aber nur dann wirksam, wenn gleichzeitig der Lightpen-Eingang LPSTRB ebenfalls auf logisch Null liegt. Der ist deshalb im CPC mit einem 10-kOhm-Widerstand "weich" nach unten gezogen, damit andererseits der Lightpen-Eingang benutzbar bleibt.

LPSTRB – LightPen STRoBe

Bei einer 0-1-Flanke an diesem Eingang wird die MA-Adresse des aktuell dargestellten Zeichens in zwei speziell dafür vorgesehenen Registern des CRTC abgespeichert. Dort kann sie von der CPU gelesen werden.

Cursor

Mit diesem Ausgang zeigt der CRTC an, daß momentan ein Zeichen auf der Cursor-Position dargestellt wird. Damit kann man einen zweiten Zeichensatz im Character-ROM adressieren oder den Pegel dieser Leitung direkt ins Monitorsignal mischen.

DISP – DISPlay Character Area

Liegt dieses Ausgangssignal auf logischem Eins-Pegel, wird ein Teil des beschreibbaren Bildschirms dargestellt. Damit signalisiert der CRTC, daß das durch ihn adressierte Byte dargestellt werden muß.

Liegt an diesem Pin jedoch ein Null-Signal an, bedeutet das, daß jetzt die Umrandung der Schreibfläche gezeichnet wird. Dann darf das Gate Array nicht die Bit-Informationen des vom CRTC (unabsichtlich) adressierten Bytes auswerten, sondern muß auf die für den BORDER programmierten Farben zurückgreifen.

CS – Chip Select

Nur wenn diese Leitung auf Null-Potential liegt, kann der CRTC mit den anderen Steuereingängen angesprochen werden. Zur Ansteuerung des CRTC im CPC erfahren Sie später mehr.

Strobe

Mit der steigenden Flanke an diesem Eingang werden die am CRTC anliegenden Steuersignale übernommen, vorausgesetzt, CS ist aktiv. Dieses Verhalten bereitet dem Z80 einige Schwierigkeiten und rührt daher, daß der HD 6845 ursprünglich für CPUs der Serie 68xx entwickelt wurde.

R/W – Read/Write-Select

Der Pegel an dieser Eingangsleitung entscheidet darüber, ob das angewählte Register durch die CPU gelesen oder beschrieben werden soll.

HSYNC – Horizontal SYNChronisation

An diesem Ausgang liefert der CRTC das Signal für die horizontale (waagerechte) Synchronisation des Monitorbildes. Damit wird die Zeilenfrequenz des Monitorbildes festgelegt.

VSYNC – Vertical SYNChronisation

Entsprechend ist dies der Ausgang für die vertikale (senkrechte) Synchronisation. Hiermit wird die Bildwiederholfrequenz festgelegt.

RS – Register Select

Mit dem Eingangssignal an diesem Pin kann die CPU auswählen, ob sie das Adreßregister (RS = 0) oder das vorher mit diesem Adreßregister ausgewählte Kontrollregister beschreiben (oder lesen) will (RS = 1).

D0 bis D7 – Data Lines

Die Datenleitungen D0 bis D7 sind direkt an den Datenbus der CPU angeschlossen. Über diese Leitungen werden die Daten in die Register des CRTC geschrieben oder aus ihnen gelesen.

MA0 bis MA13 – Memory Address Lines

Diese Leitungen werden direkt mit den entsprechenden Adreßleitungen des Bildwiederholspeichers verbunden. Mit ihnen adressiert der CRTC immer genau das Zeichen, das gerade dargestellt werden soll.

RA0 bis RA4 – Row Address Lines

Normalerweise wird das aus dem Bildwiederholspeicher ausgelesene Byte (der Code des darzustellenden Zeichens) benutzt, um dessen Zeichenmatrix im Character-ROM zu adressieren. Dabei ist dann noch zusätzlich die Auswahl der aktuellen Rasterzeile innerhalb des Zeichens erforderlich. Diese wird dann über die Adreßleitungen RA0 bis RA4 vorgenommen. Im CPC adressiert man aber mit den Row-Address-Lines RA0, RA1 und RA2 direkt den Bildwiederholspeicher, wodurch dieser hundertprozentig grafikfähig wird.

PORT-ADRESSEN DES 6845

Zum Adressieren des PSG wird die Adreßleitung A14 benutzt. Eine 0 auf dieser Leitung zusammen mit 0 an IORQ der CPU spricht den CRTC an. Dabei werden die Adreßleitungen A8 und A9 benutzt, um das Adreßregister oder das adressierte Kontrollregister im CRTC anzuwählen (A8 ist direkt mit RS verbunden) und um den CRTC auf Schreib- oder Lesezugriff einzustellen (A9 liegt an R/W an). Daraus ergeben sich folgende Adressen zur Programmierung des CRTC im Schneider CPC:

```
OUT (&BCxx)  Video-Chip-Register adressieren
OUT (&BDxx)  Video-Chip-Register beschreiben
 IN (&BExx)  reserviert für Status lesen
 IN (&BFxx)  Video-Chip-Register einlesen
```

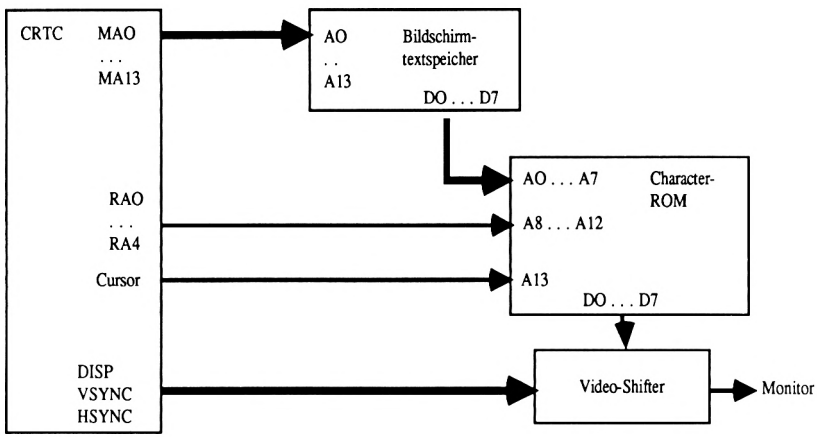
Dabei stellt sich die Frage: Wieso wird A9 benutzt, um den CRTC auf Lesen oder Schreiben umzustellen, und nicht das WR-Signal des Prozessors?

Die Antwort ist, daß der HD 6845 die Signale der CPU mit der steigenden Flanke am Strobe-Eingang übernimmt. Ein Verhalten, das sich mit dem Z80 nicht so ohne weiteres abstimmen läßt. Nun ist es den AMSTRAD-Entwicklern mit diesem Trick aber gelungen, den CRTC mit nur einem einzigen zusätzlichen NAND-Gatter an den Z80 anzuschließen!

Bei allen Buszugriffen legt der Z80 als allererstes die Adresse auf die Adreßleitungen. Erst wenn diese "stehen", folgen entsprechende Steuersignale: Bei I/O-Zugriffen aktiviert der Z80 das IORQ-Signal gleichzeitig mit WR oder RD. Wenn man nun aus dem verknüpften Signal von A14 (Selektionsadresse) und IORQ das Strobe-Signal bastelt, wozu es kaum eine Alternative gibt, kann man die WR-Leitung des Prozessors *nicht* benutzen, um den CRTC auf Schreiben oder Lesen umzustellen. Das Strobe-Signal darf ja erst kommen, nachdem am RS- und R/W-Eingang des CRTC die entsprechenden Signale anliegen, sonst wäre ein sicherer Betrieb nicht gewährleistet.

Aus diesem Grund hat man auch den R/W-Eingang an eine Adreßleitung angeschlossen, denn, wie oben erwähnt, werden diese von der Z80 als erstes gesetzt.

Normale Konfiguration des CRTC in einem Textsystem



NORMALER EINSATZ DES HD 6845

Normalerweise wird dieser Video-Controller in Textsystemen benutzt. Hierbei verwaltet er keinen Grafik- sondern einen reinen Textbildschirm. Im Bildwiederholspeicher des Computers wird nur der Code des darzustellenden Zeichens eingetragen. Die Grafikinformation über das Aussehen aller Zeichen ist in einem zusätzlichen Character-ROM fest gespeichert.

Der HD 6845 ist in seinem Zeitverhalten in sehr weiten Grenzen programmierbar. Seine Ausgangssignale hängen dabei von seiner Programmierung, aber auch von der Frequenz des Eingangstaktes ab.

Unabhängig davon, ob man ihn nun für Grafik- oder Textdarstellung einsetzt, liefert er die benötigten Synchronisationssignale und am Pin DISP ein Flag, das anzeigt, ob der Kathodenstrahl im Monitor im Moment ein Teilstück des beschreibbaren Bildschirmausschnittes zeichnet oder nicht. Gibt er an diesem Pin ein Eins-Signal aus, überstreicht der Kathodenstrahl den Textbereich des Bildschirms. Nur dann muß auf den Bildwiederholspeicher zugegriffen werden.

Auch bei reiner Textdarstellung muß letztendlich das Aussehen eines Zeichens ermittelt und dargestellt werden. Den Code eines Zeichens als gesetzte und nicht gesetzte Punkte auf dem Bildschirm darzustellen, ist vollkommen sinnlos.

Der Text-Computer muß deshalb ein ROM enthalten, in dem die Grafikinformationen für alle darstellbaren Zeichen fest gespeichert sind. Jedes Zeichen ist hier in Form einer Matrix gespeichert: Diese kann beispielsweise wie beim Schneider CPC genau 8*8 Punkte groß sein. Eine waagerechte Punktreihe würde dann genau ein Byte des ROMs beanspruchen: Ein Byte enthält 8 Bits, und jedes Bit kann gesetzt sein oder nicht, was dann den gesetzten bzw. nicht gesetzten Punkten innerhalb eines Buchstabenfeldes auf dem Monitor entspricht.

Insgesamt 8 solcher Bytes enthalten dann die komplette Information darüber, wie das Zeichen aussieht. Um an ein solches Grafik-Byte heranzukommen, muß man das ROM entsprechend adressieren: Dazu wird auf 7 oder 8 Adreßleitungen der Zeichencode angelegt, womit das Zeichen bestimmt wäre. Auf drei weiteren Adreßleitungen muß man die Nummer der Rasterzeile anlegen, um eins der 8 möglichen Bytes aus der Buchstabenmatrix auszuwählen. Darüber hinaus ist auch noch der Anschluß der Cursor-Leitung an eine weitere Adresse denkbar, um hiermit komplett auf einen zweiten Zeichensatz umzuschalten.

Sind die Zeichen in einer anders dimensionierten Matrix definiert, muß man die Anzahl der Bits pro Speicherzelle einfach der Matrixbreite anpassen. Ein Zeichen-ROM kann ja durchaus auch 10 oder 12 Bit breit sein, unabhängig von der Datenbus-Breite der CPU.

Für die Auswahl der Rasterzeile innerhalb der Zeichenmatrix kann der CRTC mit seinen Rasterzeilen-Adreßleitungen RA0 bis RA4 bis zu 5 Adreßbits erzeugen. Damit kann eine Zeichenmatrix bis zu 32 Rasterzeilen hoch sein.

Der CRTC übernimmt bei der Bilddarstellung außer den Synchronisationssignalen nur noch die Adressierung der Speicherzellen im Zeichen-ROM. Das funktioniert dann folgendermaßen:

Der CRTC adressiert mit den Leitungen MA0 bis MA13 den Bildschirmspeicher. Auf den Datenleitungen geben die Speicher-ICs dadurch den Code des darzustellenden Zeichens aus.

Die Datenleitungen werden deshalb direkt benutzt, um das Zeichen-ROM zu adressieren (sozusagen eine Hardware-mäßige indirekte Adressierung). Die Rasterzeile innerhalb des Zeichens adressiert der CRTC mit den Adreßleitun-

gen RA0 bis RA4 dann wieder selbst. Als weitere, zusätzliche Adresse ist auch noch das Cursor-Signal direkt verwendbar.

An den Datenleitungen des Zeichen-ROMs stehen damit die Pixel-Informationen bereit, von wo sie normalerweise in ein Schieberegister, den Video-Shifter, übernommen und Punkt für Punkt ausgegeben werden.

DER EINSATZ DES HD 6845 IM SCHNEIDER CPC

Im Schneider CPC ist der CRTC so beschaltet, daß der Bildschirmspeicher die darzustellende Grafik-Informationen direkt enthalten muß. Der Umweg über ein Character-ROM wird einfach weggelassen.

Alle Funktionen, um aus den vom CRTC gelieferten Signalen und den Bildschirm-Bytes das Monitorsignal zu erzeugen, sind im Gate Array zusammengefaßt. Das Cursor-Signal wird nicht benutzt.

Das Gate Array verteilt auch den Speicherzugriff zwischen CRTC und CPU. Da der Bildspeicher im normalen RAM liegt, kann der CRTC hier nicht ungestört zugreifen. Das ist aber keine grafiktypische Eigenheit, sondern könnte bei einem Textsystem ebenfalls der Fall sein.

Der CRTC wird mit einer Frequenz von 1 MHz betrieben. Dadurch legt er pro Mikrosekunde eine Adresse an das RAM an. Die so adressierte Speicherzelle kann vom Gate Array übernommen werden, um daraus, in ihrer Funktion als Video-Shifter, in der nächsten Mikrosekunde das Bildsignal für den Monitor zu erzeugen.

Eine kurze Rechnung zeigt aber, daß auf diese Weise in der zur Verfügung stehenden Zeit keine 80 Zeichen, sprich 80 Bytes im Bildschirmspeicher adressiert werden können.

Pro Sekunde werden 50 Vollbilder mit je etwa 310 Zeilen gezeichnet. Davon fallen 200 Zeilen in den Bereich des beschreibbaren Bildschirmausschnittes. Etwa ein Drittel der Bildschirmzeile zeichnet der Kathodenstrahl kein Bild, sondern den Border, oder er befindet sich im Strahlrücklauf. Die zur Verfügung stehende Zeit für eine komplette Zeile im beschreibbaren Bildschirmbereich läßt sich grob überschlagen mit

$$t = \frac{1.000.000}{50 * 310} * \frac{2}{3} = 43 \text{ Mikrosekunden}$$

Der CRTC schafft also gerade die Hälfte! Das ist aber nicht weiter schlimm: Das Gate Array hilft einfach bei der Adreßbildung mit, indem es selbst die unterste Adreßleitung A0 verwaltet: Während der CRTC seine Adresse anliegen hat, legt das Gate Array A0 auf Null-Pegel, liest das Byte aus dem Bildschirmspeicher, verändert A0 nach logisch Eins und liest das Byte aus der Speicherzelle danach. Es werden also pro Mikrosekunde 2 Bytes ausgelesen. Somit sind auch 80 Zeichen pro Zeile darstellbar.

Um die Bildschirmdarstellung grafikfähig zu machen, genügt es nicht, nur die normalen Adreßleitungen MA0 bis MA13 an den Bildschirmspeicher anzulegen; man muß auch die Rasteradressen RA0 bis RA4 benutzen. Dadurch stellt der CRTC bis zu 19 Adreßleitungen bereit. Mit A0, die vom Gate Array selbst erzeugt wird, ergeben sich sogar 20 Adreßleitungen, mit denen sich ein Bildschirmspeicher von einem ganzen Megabyte adressieren ließe!

Soweit ist man beim Schneider CPC aber nicht über's Ziel hinaus geschossen, sondern hat sich mit 15 Adreßleitungen des CRTC begnügt. Damit kann er auf das gesamte RAM zugreifen. Der Bildschirmspeicher muß beim CPC nicht unbedingt im oberen RAM-Viertel liegen, wenn er dort auch am sinnvollsten untergebracht ist.

Das RAM wird für die Bildschirmausgabe wie folgt adressiert:

RAM		Adreßquelle
A0	–	vom Gate Array
A1 bis A10	–	MA0 bis MA9
A11 bis A13	–	RA0 bis RA2
A14 und A15	–	MA12 und MA13

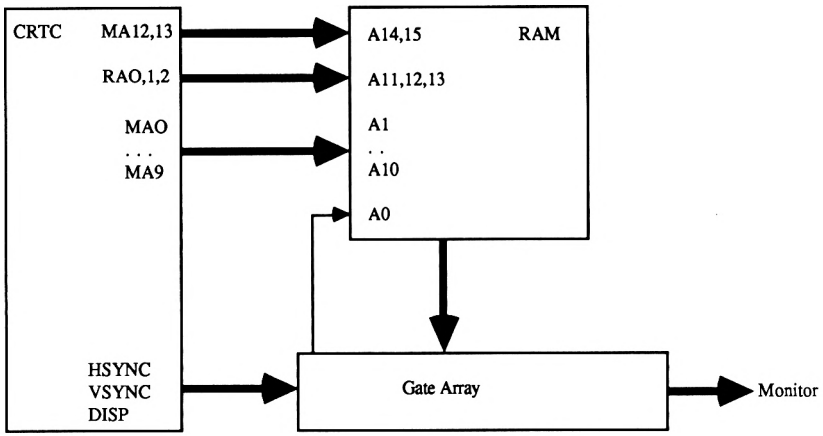
RA3 und RA4 sind nicht beschaltet. Der CRTC läßt sich aber so programmieren, daß er die Rasterzeilen-Adressen nur von 0 bis 7 durchzählt. Dann kommt man mit RA0 bis RA2 aus.

MA10 und MA11 sind ebenfalls nicht angeschlossen. Dadurch wird ein Effekt erreicht, der später noch sehr wichtig ist: Nur dadurch kann der Schneider CPC seinen Bildschirm Hardware-mäßig scrollen.

Mit MA12 und MA13 wird das Speicherviertel angewählt, in dem der Bildschirmspeicher liegen soll.

Dadurch, daß A0 nicht vom CRTC selbst erzeugt wird, kann der Bildschirmspeicher nur um jeweils zwei Bytes horizontal gescrollt werden. Das entspricht der Breite eines Buchstabens im Modus 1.

Der HD 6845 im Schneider CPC



In dieser Grafik ist nicht berücksichtigt, daß das RAM auch von der CPU adressiert wird. Dafür sind in den Adreßleitungen zum RAM Multiplexer zwischengeschaltet, die vom Gate Array gesteuert werden. Damit werden einerseits abwechselnd die Adressen des CRTC und der CPU, andererseits auch die RAS- und CAS-Adreßhälften zu den dynamischen RAMs durchgeschaltet.

DIE REGISTER DES VIDEO-CONTROLLERS HD 6845

Der HD 6845 verfügt über insgesamt 18 interne Kontrollregister, die teilweise jedoch nur beschrieben oder nur gelesen werden können. Trotzdem benötigt man normalerweise nur zwei verschiedene Adressen, um den CRTC anzusprechen. Durch die Sparschaltung, mit der man bei AMSTRAD den CRTC in das Gesamtsystem eingebunden hat, muß man im Schneider CPC allerdings vier verschiedene Adressen unterscheiden.

Um eins der sogenannten Kontrollregister zu beschreiben oder zu lesen, muß man es vorher mit dem Adreßregister anwählen. Dafür dient der Eingang RS (Register Select) des CRTC. Dieser ist direkt mit der Adreßleitung A8 der CPU verbunden. A9 hingegen liegt am Eingang R/W (Read/Write Select) des CRTC an. Die Gründe wurden ja schon weiter oben erläutert.

Daraus ergeben sich folgende Adressen, mit denen der CRTC im Schneider CPC angesprochen wird:

```
OUT (&BCxx)     Video-Chip-Register adressieren
OUT (&BDxx)     Video-Chip-Register beschreiben
 IN (&BExx)     reserviert für Status lesen
 IN (&BFxx)     Video-Chip-Register einlesen
```

Um also beispielsweise den Wert &40 ins Register 12 zu schreiben, muß man wie folgt vorgehen:

```
LD   C,12   ; Registernummer
LD   B,#BC  ; Port-Adresse CRTC-Register adressieren
OUT  (C),C  ; 12 in das Adressregister latchen
;
LD   C,#40  ; Wert für Register 12
LD   B,#BD  ; Port-Adresse CRTC-Register beschreiben
OUT  (C),C  ; #40 in das angewählte Register 12 latchen
```

Zu beachten ist dabei wieder, daß durch die spezielle I/O-Decodierung im Schneider CPC der Befehl OUT (C),C eigentlich mehr einem OUT (B),C entspricht.

Die folgende Tabelle enthält eine Aufstellung aller 18 Register des HD 6845. Zu jedem Register ist angegeben, ob es sich beschreiben bzw. ob es sich lesen läßt. Die meisten Register sind nur beschreibbar. Außerdem ist angegeben, mit welchen Werten dieses Register beim Einschalten des Computers initialisiert wird. Dabei muß jedoch zwischen PAL, SECAM und der NTSC-Norm für Amerika unterschieden werden. Die Werte für PAL und SECAM sind identisch. Für NTSC war das nicht möglich, da hier ein Bild mit einer Wiederholungsfrequenz von 60 Hz geschrieben werden muß. In Deutschland wird das Video-Signal entsprechend der PAL-Norm erzeugt.

r = lesbares / w = beschreibbares Register Werte rechts: Standardeinstellung:			NTSC dez/hex	PAL/SECAM dez/hex
R00:	(–/w)	theoretische Zeichenzahl für eine Zeile inkl. Border & Strahlrücklauf	63 &3F	63 &3F
R01:	(–/w)	dargestellte Zeichen pro Zeile	40 &28	40 &28
R02:	(–/w)	Zeitpunkt für horizontale Synchr.	46 &2E	46 &2E
R03:	(–/w)	Breite des horizontalen Synchr.-Pulses	142 &8E	142 &8E
R04:	(–/w)	theoretische Zeichenzahl für eine Spalte inkl. Border & Strahlhochlauf	31 &1F	38 &26
R05:	(–/w)	Feinabgleich zu R04	06 &06	00 &00

r = lesbares / w = beschreibbares Register Werte rechts: Standardeinstellung:			NTSC dez/hex	PAL/SECAM dez/hex
R06:	(–/w)	dargestellte Zeichen pro Spalte	25 &19	25 &19
R07:	(–/w)	Zeitpunkt für vertikale Synchr.	27 &1B	30 &1E
R08:	(–/w)	Schalter für Zeilensprung-Verfahren	00 &00	00 &00
R09:	(–/w)	Rasterzeilen/Buchstabe – 1	07 &07	07 &07
R10:	(–/w)	Einstellung für Hardware-Cursor	00 &00	00 &00
R11:	(–/w)	Einstellung für Hardware-Cursor	00 &00	00 &00
R12:	(r/w)	Text-Startadresse (msb)	48 &30	48 &30
R13:	(r/w)	Text-Startadresse (lsb)	00 &00	00 &00
R14:	(r/w)	Adresse des Hardware-Cursors (msb)	192 &C0	192 &C0
R15:	(r/w)	Adresse des Hardware-Cursors (lsb)	00 &00	00 &00
R16:	(r/)	Lightpen-Position (msb)	–	--
R17:	(r/)	Lightpen-Position (lsb)	--	--

ERKLÄRUNG ZU DEN EINZELNEN REGISTERN

R00: (–/w) theoretische Zeichenzahl für eine Zeile inkl. Border und Strahlrücklauf

Hiermit wird ein Zähler programmiert, der den Abstand zwischen zwei HSYNC-Impulsen festlegt. "Zeichenzahl" ist dabei jedoch irreführend. Normalerweise entspricht ein Speicherzugriff des CRTC auch einem Buchstaben auf dem Monitor, was beim Schneider CPC aber nicht der Fall ist. Die Bezeichnung "Speicherzugriffe" wäre vielleicht korrekter. Der korrekte Wert für PAL läßt sich leicht aus der Taktfrequenz des CRTC, der Bildfrequenz und der Zeilenzahl berechnen:

$$R00 = \frac{1.000.000}{50 * 312} = 64$$

R01: (–/w) dargestellte Zeichen pro Zeile

Auch hier ist wieder die Anzahl der Speicherzugriffe für eine Bildschirmzeile gemeint; jetzt jedoch, wie viele Adressen der CRTC tatsächlich ausgeben muß. R01 enthält also normalerweise die Anzahl der Zeichen, die in einer Zeile dargestellt werden sollen. Beim CPC entsprechen 80 Bytes einer Zeile im Bildschirmspeicher. Da der CRTC die unterste Adresse A0 nicht selbst ver-

waltet, wird R01 nur mit 40 programmiert. Wer hier andere Werte hineinschreibt, muß sich alle Routinen des Betriebssystems, die auf den Bildschirmspeicher zugreifen, neu schreiben.

R02: (–/w) Zeitpunkt für horizontale Synchronisation

Durch Verändern des Wertes in diesem Register kann man das Bild auf dem Monitor nach links oder nach rechts schieben. Verkleinert man den Wert, kommt der Synchronisationsimpuls früher. Dadurch wird die Zeitspanne zwischen Impuls und Beginn der Bilddarstellung länger, das Bild verschiebt sich also nach rechts.

R03: (–/w) Breite des horizontalen Synchronisationspulses

Von diesem Register werden nur die unteren 4 Bits benutzt. Damit wird die Breite sowohl des vertikalen als auch des horizontalen Synchronisationsimpulses festgelegt.

R04: (–/w) theoretische Zeichenzahl für eine Spalte inkl. Border und
R05: (–/w) Strahlhochlauf

Auch hier geht man wieder davon aus, daß der CRTC selbst bestimmt (programmierbar durch R09), wie hoch ein Zeichen sein soll. Zufällig stimmt der Wert jedoch auch für den Schneider CPC, da dessen Buchstaben 8 Rasterzeilen hoch sind, was den drei benutzten Raster-Adreßleitungen des CRTC entspricht, die zum Adressieren des Bildschirmspeichers benutzt werden: $2^3 = 8$. Aus dem Wert, mit dem R04 im Schneider CPC programmiert wird, läßt sich errechnen, wie viele Rasterzeilen der Schneider CPC tatsächlich zum Monitor bringt:

$38 * 8 = 304$

200 Zeilen liegen im Bereich des beschreibbaren Bildschirmausschnittes. Die restlichen 104 Zeilen verteilen sich auf untere und obere Umrandung.

R06: (–/w) dargestellte Zeichen pro Spalte

Hier gilt dasselbe wie bei R05. R06 bezieht sich nur auf den tatsächlich beschreibbaren Bildschirmausschnitt.

R07: (–/w) Zeitpunkt für vertikale Synchronisation

Mit diesem Register läßt sich der Zeitpunkt für das VSYNC-Signal festlegen. Damit kann man das Monitorbild nach unten oder nach oben verschieben. Wird der Wert in diesem Register verringert, so wird der Impuls früher erzeugt. Dadurch wird die Zeitspanne zwischen Impuls und Beginn des beschreibbaren Bildschirmbereichs länger, das Monitorbild also nach unten verschoben.

R08: (–/w) Schalter für Zeilensprung-Verfahren

In Bit 0 und 1 dieses Registers befinden sich Flags, mit denen der CRTC auf Zeilensprung-Verfahren umgeschaltet werden kann. Dann werden keine 310 Rasterzeilen mit einer Bildwiederholfrequenz von 50 Hz geschrieben, sondern 625 mit nur noch 25 Hz. Dieses Verfahren wird für Fernsehbilder verwendet. Im Schneider CPC ist es jedoch aufgrund der Beschaltung der Adreßleitungen nicht anwendbar.

R09: (–/w) maximale Rasterzeilen-Adresse

Mit diesem Register wird festgelegt, wie viele Rasterzeilen die Buchstaben hoch sein sollen. Damit werden die Row-Address-Leitungen RA0 bis RA4 des CRTC programmiert. Da diese Leitungen beim Schneider CPC jedoch direkt das RAM adressieren, ist hier kein anderer Wert als die standardmäßige 7 sinnvoll. Programmiert man einen kleineren Wert, fehlen in jeder Buchstabenzeile die untersten Rasterzeilen, und der dargestellte Bildschirmausschnitt wird schmaler. Da sich dadurch auch die Bildwiederholfrequenz ändert, müßte man auch noch R05 umprogrammieren, sonst bekommt man kein stehendes Bild mehr.

R10: (-/w) Einstellung für Hardware-Cursor

Die Bits 0 bis 4 legen fest, in welcher Rasterzeile der Cursor beginnen soll. Die Bits 5 und 6 bestimmen, wie der Cursor dargestellt wird:

Bit 5	6	Cursor
0	0	dargestellt, blinkt nicht
0	1	blinkt ca. 3mal/Sek.
1	0	kein Cursor
1	1	blinkt ca. 1,5mal/Sek.

R11: (–/w) Einstellung für Hardware-Cursor

Entsprechend R10 wird festgelegt, auf welcher Rasterzeile der Cursor endet. Mit R10 und R11 läßt sich ein Cursor in vielen Formen darstellen; das CPC-gewohnte "Inverse Patch" ebenso wie ein einfacher "Unterstrich". Leider ist er im Schneider CPC nicht einsetzbar.

R12: (r/w) Text-Startadresse
R13: (r/w)

Diese Register legen fest, ab welcher MA-Adresse der Textspeicher beginnen soll. Normalerweise wird das dazu benutzt, um das Monitorbild Hardware-mäßig durch einen größeren Textspeicher scrollen zu können (bei reiner Textdarstellung). Dadurch, daß man beim Schneider CPC die Adreßleitungen MA10 und MA11 nicht beschaltet hat, wurde erreicht, daß der CRTC, wenn er am Ende des Bildschirmspeichers ankommt, an dessen Anfang weitermacht. Der Übertrag von MA9 nach MA10 findet zwar statt, hat aber keinen Effekt bei der Adressierung des Bildschirm-RAMs. Die Adressen MA12 und MA13, mit denen das Speicherviertel für den Bildwiederholspeicher ausgewählt wird, werden davon nicht betroffen, wenn man als Bildspeicher-Startadresse Werte programmiert, bei denen MA10 und MA11 null sind. Dadurch kann man auch den Grafikbildschirm des CPC Hardware-mäßig scrollen, allerdings nicht durch einen größeren Textspeicher, sondern immer auf der Stelle. Was hinten herausscrollt, kommt vorne mit einem leichten Versatz wieder herein und muß gelöscht oder überschrieben werden.

Wenn man als Bildspeicher-Startadresse Werte programmiert, bei denen MA10 und MA11 gleich Eins sind, macht der Video-Controller mit den RAM-Adressen nicht mehr am Anfang weiter, wenn er das Ende erreicht hat. Dann geht der Übertrag von MA9 über MA10 und MA11 bis MA12 und evtl. auch MA13, und die Bildausgabe geht aus dem nächsten Speicherviertel weiter. Dadurch könnte man rein Hardware-mäßig auch beim Schneider CPC durch einen Grafikspeicher von etwa 51 Buchstabenzeilen scrollen.

Beeinträchtigt wird dieses Vergnügen nur durch den verbrauchten Speicherplatz und dadurch, daß neben dem normalen Bildschirmspeicher nur die beiden schlechter verwertbaren Speicherviertel liegen: Der unterste Block enthält leider die wichtigen Restarts, und Block 3 die Jump-Blocks und die Interrupt-Routine. Man könnte aber beispielsweise in Block 2 anfangen und in Block 3 nur bis zu den vom Betriebssystem belegten Bereichen weitermachen. Dann beschränkt sich der Bildspeicher auf etwa 45 Zeilen. Die Text- und Grafik-Routinen des CPC-Betriebssystems sind dann natürlich nur noch bedingt einsatzfähig.

R14: (r/w) **Adresse des Hardware-Cursors**
R15: (r/w)

In diese Register wird die MA-Adresse des Zeichens geschrieben, auf dem der Cursor dargestellt werden soll.

R16: (r/–) **Lightpen-Position**
R17: (r/–)

Nach einer 0-1-Flanke am Lightpen-Eingang enthält dieses Register die MA-Adresse des Zeichens, das in dem Augenblick dargestellt wurde, als dieser Impuls eintraf. Im Bildschirmmodus 1 kann man so die Position eines Lightpens auf eine Buchstabenposition genau ermitteln. Für Grafikanwendungen ist das etwas dürftig, für eine Menü-Auswahl aber sicherlich ausreichend.

R18 – R31

Das 5 Bit breite Adreßregister kann man natürlich auch noch mit den illegalen Registeradressen 18 bis 31 programmieren. Diese werden vom CRTC aber automatisch ignoriert.

Die ULA 40007, 40008 oder 40010 und das PAL HAL16L8

Die ULA (User designed Logic Array), im Schneider CPC auch oft Gate Array genannt, ist das einzige nicht serienmäßige IC im Schneider CPC.

In jedem Computer müssen eine Vielzahl von Logikgattern, Flip-Flops oder Schieberegistern für das ungestörte Zusammenspiel der einzelnen hochintegrierten Bauteile sorgen. Eine Möglichkeit ist dabei, dafür die entsprechenden serienmäßigen ICs zu benutzen, beispielsweise die der Serie 74LS. Der vergleichweise hohe Aufwand für viele einzelne ICs treibt aber unweigerlich den Preis des Computers in die Höhe. Ab bestimmten Stückzahlen lohnt es sich, die benötigten Funktionen auf einem Chip zusammenzufassen und speziell für den einen Computer herstellen zu lassen.

Ein PAL ist ein serienmäßiges IC mit einer Ansammlung von Logikgattern, die dauerhaft in ihrer Verknüpfung programmiert werden können. Es wird dann sinnvoll eingesetzt, wenn man für einen Computer zwar eine IC-Anhäufung vermeiden will, die benötigten Stückzahlen aber noch zu gering sind, um ein "eigenes" IC wirtschaftlich erscheinen zu lassen.

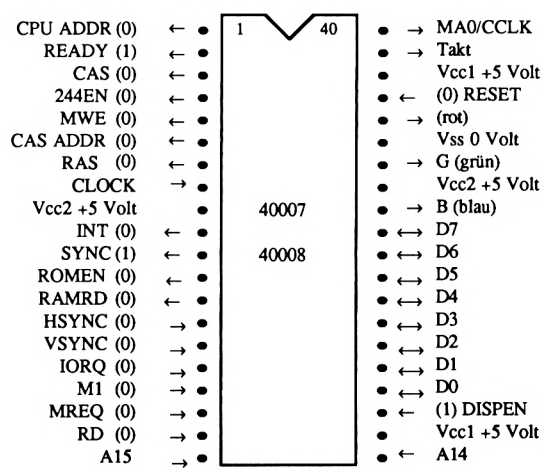
Der Unterschied zwischen PAL und ULA läßt sich mit dem zwischen einem PROM und einem ROM vergleichen:

	logische Verknüpfungen	Speicherzellen mit Daten
Inhalt des ICs wird beim Hersteller festgelegt	ULA	ROM
Inhalt wird vom Anwender programmiert	PAL	PROM

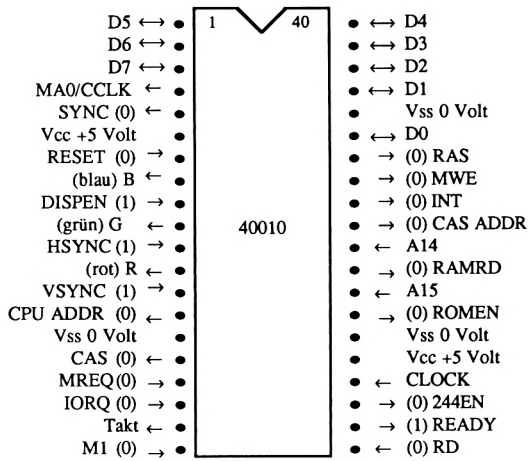
Außerdem kann das PAL auch noch durch die CPU programmiert werden. Das geschieht natürlich nicht dauerhaft, sondern in eigens dafür geformte Register hinein. Die Port-Adresse für das PAL ist dieselbe wie für das Gate Array. Bei dem wird noch eine Zusatzauswahl der angesprochenen Register über die Bits 6 und 7 des Datenbytes vorgenommen, wobei die Kombination 11 keine definierte Funktion hat. Diese Adreßlücke wird für das PAL genutzt.

Obwohl das Gate Array bei allen drei CPCs dieselben Aufgaben übernimmt, gibt es für jeden Computer eine neue Version. Für den CPC 6128 wurde die Anschlußbelegung des ICs komplett umgestaltet.

DIE ANSCHLUSSBELEGUNG DER ULA 40007 UND 40008 (CPC 464 UND 664)



DIE ANSCHLUSSBELEGUNG DER ULA 40010 (CPC 6128)



Erklärung zu den Anschlüssen:

Vcc und Vss

Über diese Anschlüsse erfolgt die Stromversorgung der ICs. Vss wird dabei an 0 Volt angeschlossen, Vdd bzw. Vcc an +5 Volt. Die Anschlüsse Vcc2 liegen dabei über einen Vorwiderstand von 6 Ohm an +5 Volt an.

RESET

Ein Null-Signal an diesem Eingang versetzt das Gate Array in den Einschaltzustand. Am wichtigsten ist hierbei, daß dadurch das untere ROM eingeblendet wird, damit die CPU überhaupt ein Initialisierungsprogramm finden kann.

CLOCK, Takt und MA0/CCLK

Fast alle Signale im Schneider CPC werden aus einem einzigen Taktsignal hergeleitet. Der Taktgenerator hat eine Frequenz von 16 MHz und speist sein Signal am Pin CLOCK in das Gate Array ein. Dort wird es durch vier geteilt und steht am Ausgang "Takt" mit einer Frequenz von nur noch 4 MHz zur Verfü-

gung. Benutzt wird es hauptsächlich als Eingangstakt für die CPU und für den FDC (Floppy Disk Controller). Deswegen ist es auch auf den Expansion-Port durchgeführt.

Noch einmal durch vier geteilt, wird das Signal mit einer Frequenz von 1 MHz am Anschluß MA0/CCLK ausgegeben. Dort dient es als Eingangstakt für den Video-Controller und den Sound-Generator. Außerdem liefert diese Leitung das niederwertigste Adreßbit A0 bei der Adressierung des Video-RAMs, mit dessen Hilfe das Gate Array bei jeder Adresse des CRTC zwei Bytes aus dem Bildwiederholspeicher holen kann.

HSYNC, VSYNC, DISPEN

Diese Eingänge sind mit den gleichnamigen Ausgängen des CRTC verbunden und liefern die drei wichtigsten Signale für den Bildaufbau.

R, G, B und SYNC

Aus den Signalen des CRTC und den mit seiner Hilfe adressierten Bytes aus dem Bildwiederholspeicher erzeugt das Gate Array diese vier Signale für einen Farbmonitor. Das von einem Grünmonitor benötigte Luminanzsignal erhält man, indem man diese vier Signale wieder zusammenmischt.

IORQ, MREQ, RD, M1, A14 und A15

Diese Eingänge sind mit den gleichnamigen Ausgängen der CPU verbunden. Hieran erkennt das Gate Array, wann die CPU lesen oder schreiben will, ob sie gerade auf den Speicher zugreift oder auf ein Peripheriegerät.

MREQ in Verbindung mit M1 zeigt der ULA nach einer Interrupt-Anforderung an, daß die CPU deren Behandlung aufgenommen hat. Dann nimmt die ULA sofort ihr INT-Signal zurück, damit die CPU in der Interrupt-Routine zwischen einem externen Interrupt und dem normalen Ticker-Interrupt des Gate Arrays unterscheiden kann.

Werden IORQ und A15 zusammen null, so deutet das darauf hin, daß die CPU gerade auf die Register des Gate Arrays zugreift, um beispielsweise eine neue Bankkonfiguration oder neue Farben zu programmieren. IORQ zeigt den I/O-Zugriff an, und A15 ist das Select-Bit für das Gate Array. Daraus resultiert die Port-Adresse &7Fxx für das Gate Array.

Werden MREQ und RD gleichzeitig null, so liest die CPU gerade Daten aus dem Speicher. Dann muß das Gate Array entsprechend der programmierten Speicherkonfiguration RAM oder ROM einblenden. Dafür muß sie das Speicherviertel erkennen, in dem der Lesezugriff stattfindet, wofür sie die Adressen A14 und A15 decodiert.

RAMRD und ROMEN

Mit diesen Ausgängen aktiviert die ULA bei einem Speicher-Schreib- oder -Lesezyklus der CPU entweder die ROMs oder schaltet die Datenleitungen von den RAM-ICs auf den Datenbus der CPU durch. Bei Schreibbefehlen wird immer das RAM aktiviert. Liest die CPU aber Daten, so erkennt die ULA anhand der Adreßleitungen A14 und A15, in welchem Speicherviertel der Zugriff erfolgt und aktiviert dann, abhängig von der in ihren Registern programmierten Bankselektion, das Signal RAMRD, wenn im betroffenen Speicherviertel RAM, oder ROMEN, wenn hier ein ROM eingeblendet werden muß.

INT – Interrupt

Dieser Ausgang ist mit dem normalen Interrupt-Eingang der CPU verbunden. 300mal in jeder Sekunde erzeugt das Gate Array hier eine Interrupt-Anforderung. Dabei werden unter anderem die Tastatur abgefragt und die blinkenden Farben in die entsprechenden Register der ULA programmiert.

READY

READY ist mit dem WAIT-Eingang des Z80 verbunden. Eigentlich sollte man der besseren Verständlichkeit wegen diesen Ausgang auch WAIT nennen. Das eins-aktive READY ist in seiner Funktion dem null-aktiven WAIT-Eingang der CPU vollkommen äquivalent. Über diesen Ausgang synchronisiert die ULA den Speicherzugriff der CPU mit dem des Video-Controllers, indem sie an diesem Ausgang nur mit jedem vierten "Takt" der CPU ein READY signalisiert.

D0 bis D7, 244EN und RAMRD

D0 bis D7 bilden den Datenbus der ULA. Dieser ist direkt mit den Datenausgängen der RAM-Bausteine verbunden, vom allgemeinen Datenbus der CPU aber durch zwei Puffer-ICs getrennt, die einen bidirektionalen Daten-Port bilden. In Richtung CPU → ULA dient ein 74LS244 als Trennschalter, der

über die Leitung 244EN vom Gate Array durchgeschaltet werden kann. Das wird gemacht, wenn die CPU ein Register im Gate Array neu programmieren will.

In der Gegenrichtung ist ein 74LS373 eingebaut. Dieses IC kann die Daten nicht nur von der ULA (bzw. den RAMs) zur CPU durchschalten, sondern auch noch speichern, wenn sich auf dem Bus der ULA bereits wieder Daten für die Bildwiedergabe befinden. Die ULA muß unabhängig von den Daten, die der Z80 gerade liest oder auf dem Datenbus ablegt, in jeder Mikrosekunde zwei Bytes aus dem Video-RAM lesen.

Bei einem Lesezyklus aus dem ROM bleibt dieser Port die ganze Zeit geschlossen. Die ROMs sind direkt an den Datenbus der CPU angeschlossen. In der Zwischenzeit kann die ULA ungestört zwei Bytes aus dem Video-RAM lesen.

Bei einem Lesezugriff auf das RAM schaltet die ULA die Adresse der CPU auf die RAM-Bausteine durch und übernimmt das Datenbyte in die Daten-Latches (Pufferspeicher) des 74LS373. Dessen Strobe-Eingang ist direkt mit dem READY-Signal zur CPU verbunden. Gleichzeitig aktiviert die ULA das Signal RAMRD, wodurch das gerade im 373 gespeicherte Byte auf den Datenbus der CPU durchgeschaltet wird. Während nun die CPU mit dem Lesen dieses Bytes beschäftigt ist, holt sich die ULA auf ihrem Bus zwei Bytes aus dem Video-RAM.

Bei einem Schreibzugriff läuft es wieder anders: Die Dateneingänge der RAM-ICs sind direkt mit dem Datenbus der CPU verbunden (die Datenausgänge nicht!).

Die CPU legt Daten und Adressen auf dem Bus ab und signalisiert mit ihren Steuerleitungen, daß sie ein Byte in eine Speicherzelle schreiben will. Das Gate Array legt den Steuereingang WE der RAM-ICs auf 0 Volt und schaltet daraufhin die Adressen der CPU auf die RAMs durch: Zuerst RAS und dann CAS. Mit CAS übernehmen die RAMs dann auch das Datenbyte von der CPU. Kaum ist das geschehen, wird die CPU auch schon wieder abgetrennt, und während diese ihren Schreibzyklus beendet, liest das Gate Array auf seinem Bus zwei Bytes für das Monitorbild.

MWE, RAS, CAS, CPU ADDR und CAS ADDR

Mit diesen Signalen wird das Beschreiben und Lesen der dynamischen RAM-Bausteine durch die ULA geregelt. MWE (Memory Write Enable) ist mit WE der RAMs verbunden und unterscheidet bei jedem Zugriff, ob diese gelesen (1) oder beschrieben (0) werden sollen.

RAS und CAS sind an die gleichnamigen Pins der RAM-ICs angeschlossen. Die dynamischen RAMs benötigen jede Adresse nämlich in zwei Hälften, einer Zeilenadresse (Row) und einer Spaltenadresse (Column). Zuerst benötigen sie die Zeile: Diese wird (von der ULA gesteuert) auf die Adreßanschlüsse der RAMs durchgeschaltet, und dann wird das RAS-Signal aktiviert (RAS – Row Address Strobe). Danach folgt die Spaltennummer. Sobald diese Adreßhälfte anliegt, aktiviert die ULA das CAS-Signal (CAS – Column Address Strobe). Mit dem CAS-Signal übernehmen die RAM-ICs die Daten vom Datenbus (bei Schreibzugriffen) oder geben die in ihnen gespeicherten Informationen aus.

Natürlich langt es nicht, den RAM-Bausteinen zu signalisieren: So, jetzt könnt ihr die Zeilen- bzw. Spaltenadresse übernehmen. Man muß diese auch an ihre Adreßleitungen anlegen. Dazu dienen die beiden Ausgänge CPU ADDR und CAS ADDR, mit denen insgesamt vier Multiplexer-ICs (Umschalter) angesteuert werden. An den Eingängen dieser Umschalter liegen einerseits die Adreßleitungen der CPU und andererseits die Adreßleitungen des CRTC an. Beide müssen bzw. können ja auf das gesamte RAM zugreifen. Mit CPU ADDR wählt die ULA zunächst einmal aus, ob die Adressen von der CPU (0) oder vom CRTC (1) durchgeschaltet werden sollen. CAS ADDR unterscheidet entsprechend zwischen der oberen Adreßhälfte (1) und der unteren (0).

Dabei werden insgesamt vier Gruppen zu je 8 Leitungen auf die 8 Adreßanschlüsse der dynamischen RAMs durchgeschaltet:

CPU ADDR	CAS ADDR	durchgeschaltet werden:
0	0	A0 bis A7 der CPU
0	1	A8 bis A15 der CPU
1	0	untere Adreßhälfte vom CRTC
1	1	obere Adreßhälfte vom CRTC

DIE REGISTER DES GATE ARRAYS UND DES PALS

Das Gate Array und beim CPC 6128 das PAL enthalten Register, die die Bildschirmdarstellung und die Speicherkonfiguration beeinflussen. Durch Programmieren dieser Register kann man festlegen, welcher Bildschirmmodus dargestellt wird, welche Farben die einzelnen Tinten haben, ob oben oder unten ein ROM eingeblendet werden soll und, beim CPC 6128, wie die zusätzlichen 64 kByte RAM verwendet werden.

ULA und PAL werden beide durch eine Null auf der Adreßleitung A15 (zusammen mit IORQ) angesprochen. Daraus ergibt sich die Port-Adresse hex: &7Fxx (binär: &X0111111xxxxxxxx). Alle Register in ULA und PAL

sind nicht lesbar, sondern können nur beschrieben werden. Zur Unterscheidung zwischen diesen beiden ICs und zur Auswahl eines bestimmten Registers in der ULA dienen keine zusätzlichen Adreßleitungen, sondern die Bits 6 und 7 des übergebenen Datenwortes.

Portadresse: &7FFF (nur Output)

1. Gate Array:

Bit 76543210

Datenwort binär	Bedeutung der einzelnen Bits im Datenbyte
&X0000iiii	Wählt das Farbregister für Tinte iiii an
&X0001????	Wählt das BORDER-Farbregister an
&X010nnnnn	Programmiert das gerade angewählte Farbregister mit der Farbe nnnnn (Paletten-Farbnummer)
&X100roumm	r=1 – Löscht den 52-Bildschirmzeilen-Zähler (Interrupt-Verzögerung) o=1 – Blendet oberes ROM aus u=1 – Blendet unteres ROM aus mm – Bestimmt Bildschirmmodus

2. PAL:

Steuerung der RAM-Bankauswahl wie folgt.
(n)ormale RAM-Bank – (z)usätzliches RAM
Bit 76543210 0,1,2,3: Block der jeweiligen Bank

Datenwort binär	CPU-Adreßviertel -0- -1- -2- -3-	Konfiguration wird verwendet bei:	normale Lage für Video-RAM
&X11000000	n0 - n1 - n2 - n3	Normalzustand	n3=Screen
&X11000001	n0 - n1 - n2 - z3	CP/M: BIOS, BDOS etc.	n1=Screen
&X11000010	z0 - z1 - z2 - z3	CP/M: TPA	(n1=Screen)
&X11000011	n0 - n3 - n2 - z3	CP/M: n3=Hush-Tabelle	(n1=Screen)
&X11000100	n0 - z0 - n2 - n3	Bankmanager	n3=Screen
&X11000101	n0 - z1 - n2 - n3	Bankmanager	n3=Screen
&X11000110	n0 - z2 - n2 - n3	Bankmanager	n3=Screen
&X11000111	n0 - z3 - n2 - n3	Bankmanager	n3=Screen

Anmerkung: Bit 5 ist immer 0 (reserviert). Das Z80-Registerpaar B'C' enthält ständig die Port-Adresse und das passende Datenbyte, um mit OUT (C),C die momentane Bankkonfiguration und den Bildschirm-Modus einzustellen.

Datenbyte &X000biiii und &X010nnnnn

Die Farbdarstellung des Schneider CPC unterscheidet Tinten (Inks) und zugehörige Farben (Colours). Im Bildschirmspeicher wird zu jedem Punkt des Bildschirms eine Tintennummer gespeichert. Während der Bildausgabe erzeugt das Gate Array daraus die zugehörige Farbe. Diese Zuordnung ist nicht fest, sondern programmierbar. Dafür benutzt die ULA eine sogenannte CLUT (Color Look Up Table), eine Tabelle, in der die Farbzuordnungen gespeichert sind. Beim Schneider CPC wird die CLUT meist als Palette bezeichnet. Sie ist in Registern in der ULA gespeichert und kann von der CPU programmiert werden.

Um die Farbzuordnung einer Tinte (Ink) zu ändern, muß diese zunächst einmal adressiert werden. Das erkennt die ULA daran, daß die beiden obersten Datenbits beide nicht gesetzt sind (Kombination 00). Unabhängig vom dargestellten Bildschirmmodus können alle 16 Tintennummern (0 bis 15) so programmiert werden. Ist auch Bit 4 gesetzt (Tintennummern von 16 bis 31), so wird das Border-Farbregister angewählt, das wie eine eigene Tinte funktioniert. Danach schreibt man den gewünschten Farbwert in das so angewählte Palettenregister. Diese Funktion wird durch die Kombination 01 der beiden obersten Datenbits angewählt.

Man muß aber aufpassen: Die Farbnummer, mit der die Palette in der ULA programmiert wird, entspricht *nicht* der Farbnummer, die man beispielsweise im INK-Statement in BASIC angeben muß. Diese Farbnummern sind nach ihren Grauwerten sortiert und werden vom Screen Pack über eine Tabelle in die entsprechenden Paletten-Farbnummern übersetzt.

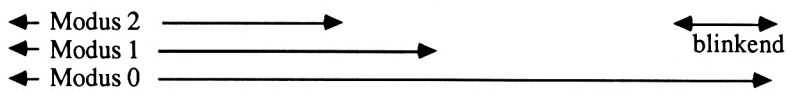
Zusammenhang zwischen Farbe, Farbnummer und Paletten-Farbnummer

Farbe	Paletten-Farbnr.	Farb-nummer	Farb-nummer	Paletten-Farbnr.	Farbe
Schwarz	20	0	26	11	Hellweiß
Blau	4	1	25	3	Pastellgelb
Hellblau	21	2	24	10	Hellgelb
Rot	28	3	23	27	Pastellblaugrün
Magenta	24	4	22	25	Pastellgrün
Hellviolett	29	5	21	26	Limonengrün
Hellrot	12	6	20	19	Hellblaugrün
Purpur	5	7	19	2	Seegrün
Hellmagenta	13	8	18	18	Hellgrün
Grün	22	9	17	15	Pastellmagenta
Blaugrün	6	10	16	7	Rosa
Himmelblau	23	11	15	14	Orange
Gelb	30	12	14	31	Pastellblau
Weiß	0	13	13	0	Weiß

Das Farb-Blinken wird vom Screen Pack Software-mäßig realisiert. Dazu hängt es einen Event-Block auf dem Frame Flyback Ticker ein, durch den im Speed-Ink-Rhythmus die Farbzuordnungen für Border und Inks immer wieder umprogrammiert werden. Bei nicht blinkenden Tinten wird dabei einfach zwischen zwei gleichen Farben "gewechselt".

Wird das Screen Pack initialisiert (beim Einschalten des Rechners beispielsweise), so nimmt es folgende Zuordnung von Farben zu den 16 Tinten und Border vor:

Tinte	Border	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Farbe 1:	1	1	24	20	6	26	0	2	8	10	12	14	16	18	22	1	16
Farbe 2:	1	1	24	20	6	26	0	2	8	10	12	14	16	18	22	24	11



Da der Bildschirm zuerst im Modus 1 dargestellt wird, können nur die Tinten 0 bis 3 angesprochen werden. Trotzdem werden bei jedem Farbblinken alle 16 Tinten neu "gefärbt". Schaltet man auf Modus 0 um, so kann man alle 16 Tinten ansprechen. Daß hier, wie in der Tabelle oben angegeben, die letzten beiden Inks blinken, davon kann man sich überzeugen, wenn man im Modus 0 PEN 15 und PEN 14 eingibt.

Die ULA kann auch von BASIC aus via OUT-Statement programmiert werden. Speziell bei den Farbzuordnungen wird man aber nicht lange Freude haben, da diese ja, wie bereits gesagt, vom Screen Pack in regelmäßigen Abständen neu programmiert werden. Wenn man aber vorher

```
SPEED INK 255,255 [ENTER]
```

eingibt, kann man den Effekt solcher OUT-Statements schon länger bewundern:

```
OUT &7FFF,&X00010000 : OUT &7FFF,&X01000111 [ENTER]
```

Das erste OUT wählt das Border-Tintenregister an, und das zweite programmiert es mit der Paletten-Farbnummer 7 in Rosa. Danach ist für eine unbestimmte Zeit (maximal 5 Sekunden) der Border in seiner neuen Farbe zu sehen, bis das nächste Farbblinken den alten Zustand wiederherstellt.

Datenbyte &X100roumm

Wird die ULA mit einem Datenbyte programmiert, in dem die beiden oberen Bits in der Kombination 10 gesetzt sind, so haben die Bits 0 bis 4 spezielle Funktionen.

Die Bits 0 und 1 zusammen bestimmen den Modus, in dem der Bildschirm dargestellt werden soll:

00	Modus 0	16 Farben	20 Zeichen pro Zeile
01	Modus 1	4 Farben	40 Zeichen pro Zeile
10	Modus 2	2 Farben	80 Zeichen pro Zeile
11	nicht spezifiziert (Modus 0)		

Ist Bit 2 gesetzt, so wird das untere ROM ausgeblendet, bei Bit 3 das obere. Sind diese Bits zurückgesetzt, so wird das entsprechende ROM eingeblendet.

Bit 2 = 0 – unten ROM
Bit 2 = 1 – unten RAM

Bit 3 = 0 – oben ROM
Bit 3 = 1 – oben RAM

Der ROM-Status bezieht sich nur auf Lesezugriffe der CPU und wird von der ULA dadurch realisiert, daß sie entsprechend ihrer Programmierung ROMEN oder RAMRD aktiviert. Schreibbefehle gehen immer an das interne RAM.

Ist Bit 4 gesetzt, so wird der Rasterzeilen-Zähler in der ULA zurückgesetzt. Dieser Zähler erzeugt, sobald er den Wert 52 erreicht, einen Interrupt. Er wird außerdem noch mit dem Strahlhochlauf synchronisiert, so daß pro Sekunde exakt 300 Interrupts erzeugt werden.

Auch dieses Register eignet sich nur sehr beschränkt für Spielereien in BASIC. Da hierin auch der ROM-Status programmiert wird, wird es bei jedem Interrupt beschrieben. Die Interrupt-Routine muß nämlich das Betriebssystem-ROM einblenden, um dort Unterprogramme ausführen zu können.

Dazu ist ständig im B'C'-Register der CPU der korrekte Wert gespeichert, um mit "OUT (C),C" den aktuellen ROM-Status des Hauptprogramms wiederherzustellen. B' enthält das höherwertige, signifikante Byte der ULA-Adresse (&7F) und C' das Datenbyte mit ROM-Status und Bildschirmmodus. Die Bits 6 und 7 sind natürlich in der Kombination 10 gesetzt.

Von BASIC aus sind deshalb kaum sinnvolle Anwendungen möglich. Man kann den CPC beispielsweise sehr effektiv in einer Schleife aufhängen, in der der Interrupt-Zähler immer wieder genullt wird. Da kein Interrupt mehr erzeugt wird, unterbleibt auch die Tastaturabfrage, und der Rechner kann nur noch durch Ausschalten wieder zum Leben erweckt werden:

```
10 OUT &7FFF,&X10010101 : GOTO 10
```

Auch ein unterhaltsames Quiz ist möglich:

```
OUT &7FFF,&X10001001
```

schaltet das obere ROM aus und das untere ein. Das ist genau der umgekehrte ROM-Status, wie ihn BASIC benötigt. Die unerwartetsten Meldungen sind so möglich: Vom einfachen "Ready" über "press play on tape" bis zum totalen Systemabsturz.

Datenbyte &X11000ram

Ein OUT auf der Adresse der ULA, bei der im Datenbyte die beiden obersten Bits gesetzt sind (Kombination 11), spricht kein Register in der ULA an. Im Schneider CPC 6128 wird hiermit aber das PAL programmiert, das die RAM-Konfiguration steuert.

Das PAL wertet nur die untersten drei Datenbits aus. Entsprechend sind acht verschiedene RAM-Konfigurationen möglich. Die nicht benutzten Bits sollte man immer auf Null setzen, falls AMSTRAD hier in Zukunft noch andere Funktionen plant. Das PAL steuert nur den Zugriff der CPU auf das RAM. ROM-Konfiguration und Bildausgabe bleiben davon unberührt. Die folgende Tabelle zeigt noch einmal die verschiedenen Möglichkeiten:

Programmierbare RAM-Konfigurationen

&X11000000	n0 - n1 - n2 - n3	Normalzustand	n3=Screen
&X11000001	n0 - n1 - n2 - z3	CP/M: BIOS, BDOS etc.	n1=Screen
&X11000010	z0 - z1 - z2 - z3	CP/M: TPA	(n1=Screen)
&X11000011	n0 - n3 - n2 - z3	CP/M: n3=Hush-Tabelle	(n1=Screen)
&X11000100	n0 - z0 - n2 - n3	Bankmanager	n3=Screen
&X11000101	n0 - z1 - n2 - n3	Bankmanager	n3=Screen
&X11000110	n0 - z2 - n2 - n3	Bankmanager	n3=Screen
&X11000111	n0 - z3 - n2 - n3	Bankmanager	n3=Screen

n=normale RAM-Bank, z=zusätzliche RAM-Bank

Wenn man den RAM-Block, der gerade den Bildwiederholspeicher enthält, ausblendet, schreibt die CPU ab sofort in einen Block des zusätzlichen RAMs, der aber nicht dargestellt wird:

```
OUT &7FFF,&X11000001 : REM n0 - n1 - n2 - z3
```

Alle Textausgaben bleiben dann unsichtbar. Trotzdem wird der sichtbare Bildschirm gescrollt, wenn der (unsichtbare) Bildschirm Hardware-mäßig gescrollt werden muß. Die CPU löscht dabei die neu entstehenden Zeilen in der zusätzlichen RAM-Bank, nicht aber in der tatsächlich dargestellten. Dadurch rollt hier die Anzeige oben heraus und unten wieder versetzt herein oder umgekehrt. Der Versatz kommt daher, daß aus einer "RAM-Zeile" des Video-RAMs, die 2k = 2048 Bytes umfaßt, nur 2000 tatsächlich benutzt werden.

Um die normale RAM-Konfiguration wieder herzustellen, muß man nun blind und fehlerfrei OUT &7FFF,&X11000000 eingeben oder den Rechner mit CTRL-SHIFT-ESCAPE neu initialisieren. Danach kann man sich mit dem Bankmanager ansehen, was man die ganze Zeit blind eingetippt hat: |SCREEN-SWAP,1,5 bringt den nicht darstellbaren dritten Block der zusätzlichen RAM-Bank, in den man die ganze Zeit geschrieben hat, in den Bildschirmspeicher.

Der FDC 765

Dieses hochintegrierte IC übernimmt im Schneider CPC die meiste Arbeit beim Datentransfer von und zur Floppy. Der FDC (Floppy Disk Controller) 765 ist dabei trotz seiner Leistungsfähigkeit wieder ein erstaunlich preiswertes und auch weit verbreitetes IC. Tatsächlich ist es einer der intelligentesten seiner Art und läßt sich problemlos mit fast jedem Mikroprozessor betreiben. Auch er benötigt nur eine einzige Stromversorgung mit 5 Volt, für seine unterschiedlichen Timing-Funktionen jedoch zwei Eingangstakte. Trotz der umfangreichen Programmierbarkeit haben es seine Entwickler geschafft, mit nur einer zusätzlichen Adreßleitung auszukommen.

Der FDC ist neben dem AMSDOS-Programm-ROM das wichtigste Bauteil im Controller für die 3-Zoll-Laufwerke. Im Schneider CPC 664 und 6128 ist er (wie das Floppy-Laufwerk) bereits im Computer eingebaut. Beim CPC 464 wird der Controller hinten am Systembus angesteckt und damit praktisch gleichwertig in das Gesamtsystem integriert.

Der FDC 765 kann mit fast jedem Diskettenlaufwerk betrieben werden: wahlweise ein- oder zweiseitig, 8 Zoll, 5.25, 3.5 oder 3 Zoll, mit oder ohne Kopf-

lade-Einrichtung, 40, 80 oder sogar noch mehr Spuren. Bis zu 4 verschiedene Laufwerke sind gleichzeitig anschließbar, und einen Shugart-kompatiblen Anschluß zu realisieren, ist kein Problem.

Entsprechend programmiert, übernimmt er den größten Teil der Arbeit beim Datentransfer. Zur Seite steht ihm dabei noch ein Datenseparator, der den vom FDC kommenden Bitstrom in schreibfertige Impulse und, beim Lesen, diesen wieder zurück in einen Bitstrom verwandelt. Beim CPC 464 handelt es sich um einen SMC 9229; bei den 612-Typen hat man sich mit dessen kleinerem Bruder, dem SMC 9216 begnügt.

Von den drei möglichen Betriebsmodi hat man sich beim Schneider CPC für den billigsten entschieden. Die schnellste aber auch teuerste Methode wäre der DMA-Betrieb. DMA heißt Direct Memory Access und bedeutet, daß Daten direkt, ohne den Umweg über die CPU, vom Controller in den Speicher transferiert werden. Dafür braucht man aber ein Spezial-IC, das diesen Prozeß überwacht, eben einen DMA-Controller. Der ist beim Schneider CPC nicht vorhanden.

Mittelmäßig ist die Methode, die CPU immer mit einer Interrupt-Anforderung zu wecken, wenn der FDC ein neues Byte komplett von der Diskette gelesen hat. Die CPU muß dann in der Interrupt-Routine das Byte abholen und selbst in die richtige Speicherzelle laden. Diese Methode schied beim Schneider aber wohl wegen dessen umständlicher Interrupt-Behandlung aus.

Blieb nur noch das Polling. Die CPU muß also regelmäßig nachschauen, ob der FDC ein Byte komplett zur Floppy geschickt hat und jetzt das nächste benötigt. Da es bei der Datenübertragung von und zur Floppy Schlag auf Schlag geht, kann die CPU in der Zwischenzeit natürlich nichts anderes tun als nachschauen, warten, nachschauen, speichern und so weiter, bis die Übertragung zu Ende ist.

Das verändert aber nicht die hohe Übertragungsgeschwindigkeit. Das System steht zwar ansonsten still, dafür fließt aber der Datenstrom zwischen Computer und Diskettenstation.

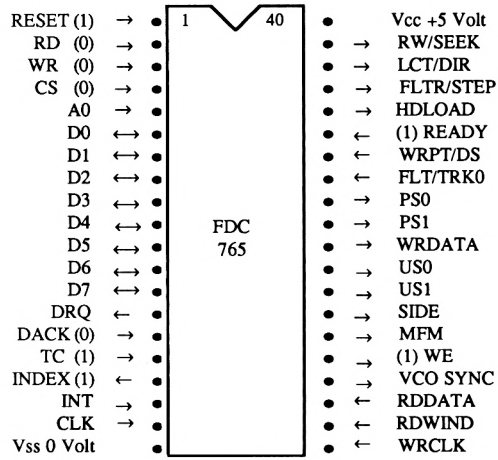
Beschränkt werden die Einsatzmöglichkeiten eigentlich eher durch ein paar kleine Spielereien, die AMSTRAD um ihn herum vorgenommen hat: So ist der FDC 765 normalerweise in der Lage, bis zu vier Laufwerke anzusteuern. Im Schneider CPC wurde diese Anzahl aber durch ein kleines Gatter künstlich auf zwei reduziert.

Auch die Möglichkeit, zweiseitige Laufwerke oder solche mit 80 Spuren anzuschließen, ist dem FDC durch AMSTRAD verwehrt worden. Diesmal

sind die Hindernisse auf der Software-Seite zu suchen. In AMSDOS sind einfach keine anderen Diskettenparameter einstellbar als einseitig, 40 Spuren und doppelte Schreibdichte.

Wer Zeit und Muße hat, sich ein eigenes DOS (Disk Operating System) zu schreiben, kann das jedoch durchaus machen.

DIE ANSCHLUSSBELEGUNG DES FCD 765



Erklärungen zu den Anschlüssen

Vss und Vcc

Als Stromversorgung benötigt der FDC 765 wie alle anderen ICs nur +5 Volt und einen GND-Anschluß (Vss).

RESET

Wie bei der PIO 8255 ist der Reset-Eingang des FDC eins-aktiv. Im normalen Betrieb liegt dieser Pin dann auf Null-Potential.

RD, WR und CS

Wenn die CPU auf den FDC zugreifen will, muß CS (Chip Select) auf Null gelegt werden. Nur dann sind auch die Steuersignale RD (Read) und WR (Write) gültig. Mit diesen Eingängen wird dem FDC mitgeteilt, ob der nächste Datentransfer in Schreib- oder Leserichtung erfolgen soll. Im allgemeinen können RD und WR direkt mit den entsprechenden Ausgängen der CPU verbunden werden.

A0 – Adresse 0

Diese einzige Adreßleitung des FDC 765 dient dazu, bei einem Schreib- oder Lesezugriff der CPU zwischen dem Haupt-Statusregister (0) und den Datenregistern (1) zu unterscheiden. Eine Unterscheidung zwischen den einzelnen Datenregistern erfolgt dabei ausschließlich durch die vom programmierten Befehl vorgegebene Reihenfolge.

D0 bis D7 – Datenbus

Über diese acht Datenleitungen erfolgt der gesamte Informationsaustausch zwischen Prozessor, evtl. auch DMA-Controller und FDC. Hier werden die Befehlsbytes und deren Parameter eingeschrieben und Statusinformationen gelesen. Hierüber "rollen" die Daten von und zur Floppy. Die Anschlüsse D0 bis D7 werden deshalb immer mit dem Datenbus der CPU verbunden.

INT – Interrupt

Der FDC 765 bietet die Möglichkeit, den gesamten Datentransfer Interruptgesteuert durchzuführen. Jedesmal, wenn der FDC ein Byte empfangen oder komplett gesendet hat, erzeugt er ein Signal an diesem Pin, der mit dem Interrupt-Eingang der CPU verbunden werden kann (im CPC nicht genutzt).

DRQ und DACK – DMA Request und DMA Acknowledge

Eine weitere (im CPC nicht genutzte) Betriebsart ist die Datenübertragung mit Hilfe eines DMA-Controllers. Dabei wird für den Datentransfer RAM → FDC die CPU nicht nur nicht benutzt, sondern sogar kurzzeitig vom Systembus abgekoppelt. Dann adressiert der DMA-Controller die Speicherzelle, und der FDC kann das Byte direkt über den Datenbus lesen oder schreiben.

TC – Terminal Count

Mit einem Eins-Signal an diesem Pin wird die Datenübertragung vom und zum FDC unterbrochen. Wird ein Schreib- oder Lesekommando gestartet, so liest oder schreibt der FDC so lange Sektoren, bis ein TC-Impuls eintrifft oder ein Fehler auftritt.

Im Schneider CPC ist TC aber einfach mit RESET verbunden. Dadurch wird dieser "terminierende" Impuls ausschließlich beim Einschalten des Computers erzeugt.

Daher enden fast alle Operationen des FDC mit einer Fehlermeldung! AMSDOS muß dann zwischen den tatsächlichen und diesem gezwungenermaßen in Kauf genommenen Fehler unterscheiden. Das ist zum Glück recht einfach: Bei jeder Schreib- oder Leseoperation muß nicht nur die Nummer des Sektors angegeben werden, mit dem man anfangen will, sondern auch die höchste Sektornummer des Tracks.

AMSDOS setzt für beide Nummern den gleichen Wert ein: Der FDC schreibt (liest) den einen Track, erhält keinen TC-Impuls und will mit dem nächsten weitermachen. Dann erkennt er aber, daß er damit die höchste Sektornummer überschreitet, und meldet: Befehl abgebrochen, weil ein Zugriff mit einer unzulässig hohen Sektornummer versucht wurde.

HDLOAD – Head Load

Dieses Signal wird meist nur bei 8-Zoll-Laufwerken benötigt und ist im AMSDOS- Controller nicht angeschlossen. Diese Laufwerke laufen im Betrieb andauernd und werden nicht, wie beispielsweise die 3-Zoll-Floppies am Schneider CPC, nur bei Bedarf gestartet. Um dabei den Schreib-/Lesekopf zu schonen, wird dieser mit einem Hubmagneten von der Diskettenoberfläche abgehoben oder darauf abgesenkt. Das ist vergleichbar mit dem Ein- und Ausschalten des Laufwerkmotors, geht aber viel schneller.

US0 und US1 – Unit Select

Über diese beiden Leitungen kann der FDC bis zu 4 verschiedene Laufwerke anwählen. Im Schneider CPC ist aber nur US0 beschaltet und liefert das Select-Signal für Laufwerk 1. Die zweite Leitung für Drive 2 ist nicht an US1 angeschlossen, sondern über einen Inverter ebenfalls an US0. Bei einer Beschränkung auf zwei Laufwerke ist das eine durchaus sinnvolle Sache, da dadurch die beiden Laufwerke nicht (verbotenerweise) gleichzeitig angesprochen werden können.

SIDE – Head Select

Bei Doppelkopflaufwerken, die beide Diskettenseiten gleichzeitig lesen und beschreiben können, wird mit dieser Leitung die obere (0) oder die untere Seite (1) angewählt. Rein Hardware-mäßig wird im AMSDOS-Controller dieser Ausgang unterstützt: Er ist zum Floppy-Anschluß durchgeführt. Die Treibersoftware im AMSDOS-ROM unterstützt aber keine Doppelkopf-Laufwerke.

INDEX

Der FDC 765 ist für "soft sektorierte" Disketten vorgesehen. Hierbei ist auf der sich drehenden Scheibe in der Diskettenhülle ein Loch, dessen regelmäßige Wiederkehr eine komplette Umdrehung anzeigt und mit einer Lichtschranke im Laufwerk abgetastet wird. Dieses Signal wird mit dem INDEX-Eingang verbunden und zeigt dem FDC an, wann die kreisförmige Spur "anfängt". Daraufhin darf er die Spur in so viele und so große Sektoren unterteilen, wie er in einer Diskettenumdrehung unterbringt (deswegen "soft" – weich).

WE – Write Enable

Mit einem Eins-Pegel an diesem Pin signalisiert der FDC dem Floppy-Laufwerk, daß er jetzt Daten schreibt.

READY

Mit dieser Leitung signalisiert das Laufwerk, daß es für einen Datentransfer bereit ist. Dazu gehört zunächst einmal, daß eine Diskette im Laufwerk eingelegt, dieses evtl. verriegelt ist und daß sich der Laufwerkmotor dreht.

WRDATA und RDDATA – Write Data und Read Data

Über diese beiden Pins läuft der serielle Bitstrom von und zur Floppy. Das geschieht jedoch nicht direkt. Hier kommt der Datenseparator ins Spiel, der beim Schreiben und Formatieren aus den Bits (0- oder 1-Pegel) und einem Taktsignal die Impulse erzeugt, die auf der magnetisierbaren Platte aufgezeichnet werden können. In umgekehrter Richtung erzeugt er beim Lesen der Diskette aus den eintreffenden Impulsen für den FDC wieder verwertbare Spannungspegel, die dieser über den Eingang RDDATA einliest.

MFM – MFM Mode Select

Über diesen Ausgang teilt der FDC dem Datenseparator mit, ob die Diskette im MFM-Modus (doppelte Datendichte) oder nur im MF-Modus (einfache Dichte) beschrieben werden soll. AMSDOS arbeitet nur mit MFM.

VCO SYNC – VCO-Synchronisation

Über diesen Ausgang kann der FDC einen spannungsgesteuerten Oszillator (VCO) im Datenseparator synchronisieren. Das wird im Schneider CPC nicht benötigt.

PS1 und PS0 – Pre Shift early und late

Um Signalverformungen der letztendlich doch analogen Aufzeichnung auf der magnetisierbaren "Floppy Disk" auszugleichen, kann man beim (empfindlicheren) MFM-Verfahren bereits beim Aufzeichnen der Daten diese Fehler vorab kompensieren. Dazu dienen die Ausgänge PS0 und PS1, mit denen der FDC dem Datenseparator das gewünschte Verfahren mitteilt.

RDWIND – Read Data Window

Hier handelt es sich um einen Signaleingang vom Datenseparator, mit dem dieser Lesedaten kennzeichnet.

RW/SEEK – Read/Write/Seek

Mit diesem Ausgang schaltet der FDC die folgenden vier Signalleitungen zwischen jeweils zwei verschiedenen Funktionen um. Damit kommt der 765 dem Shugart-Bus sehr entgegen, bei dem man mit diesem Trick vier Adern im Verbindungskabel gespart hat. Ein Null-Pegel an diesem Ausgang signalisiert Schreib-/Lese-Funktionen (RW), eine Eins Spursuchen (Seek). Die Bezeichnungen der folgenden Leitungen enthalten jeweils links die RW-Bedeutung und rechts die für SEEK.

FLT/TRK0 – Fault/Track 0

Fault (1): Manche Floppy-Laufwerke verfügen über ein Fehler-Flip-Flop, das sie setzen, sobald irgendein nicht weiter spezifizierter Fehler auftritt. Dieses

Flag wird vom FDC nach jedem Schreib- oder Lesezugriff überprüft. Im AMSDOS-Controller wird aber durch einen Schaltungstrick hier immer das Signal "Kein Fehler" erzeugt.

Track 0: Dies ist das einzige Leitung, über die der Controller jemals erfahren kann, in welcher Spur sich der Schreib-/Lesekopf eigentlich befindet. Dieses Signal wird vom Laufwerk mit einer Lichtschranke oder einem Schalter erzeugt, wenn der Kopf in der physikalisch ersten Spur steht. Die Kenntnis über die Lage des Kopfes in jeder anderen Spur erhält der FDC nur dadurch, daß er die Impulse für den Schrittmotor selbst mitzählt.

FLTR/STEP – Fault Reset/Step

Fault Reset: Nach einem Fehler (s.o.) kann der FDC mit diesem Signal das Flip-Flop wieder zurücksetzen. Im AMSDOS-Controller wird diese Funktion durch ein Gatter mit jedem Null-Pegel an RW/SEEK (Funktion: RW) erzeugt.

STEP: Mit jedem Impuls, den der FDC an diesem Ausgang erzeugt, wird der Schreib-/Lesekopf um eine Spur weitergestellt.

LCT/DIR – Low Current/Direction

Low Current: Der FDC aktiviert diese Leitung, um beim Schreiben auf die inneren Spuren der Diskette die Signalamplitude etwas zu verringern.

Direction: Diese Leitung gibt in Verbindung mit STEP an, in welche Richtung der Schritt erfolgen soll.

WRPT/DS – Write Protect/Double Sided

Write Protect: Dieses Signal wird von einer Lichtschranke im Floppy-Laufwerk geliefert, die das Schreibschutzloch durchleuchtet. Erkennt der FDC, daß die Diskette schreibgeschützt ist, so weigert er sich, irgendwelche Daten zu schreiben oder die Diskette zu formatieren.

Double Sided: Doppelkopf-Laufwerke können sich als solche zu erkennen geben, indem sie dieses Flag setzen.

ANSTEUERUNG DES DISKETTEN-CONTROLLERS

Das gesamte Floppy-Interface (außer dem AMSDOS-ROM) wird durch einen Peripheriezugriff der CPU (IORQ=0) angesprochen, bei dem die Adreßbits

A10 und A7 auf Null-Pegel liegen. A10 signalisiert dabei den Zugriff auf ein Gerät, das am Expansion-Port angeschlossen ist (auch bei den CPCs 664 und 6128 ist der Floppy-Controller logisch ein "externes" Gerät), und A7 selektiert das Diskettenlaufwerk. Zur weiteren Funktionsauswahl werden noch A0 und A8 benutzt.

Liegt bei einem OUT-Befehl auch A8 auf Null, so ist nicht der FDC angesprochen, sondern ein Flip-Flop, dessen Ausgang das Motor-ON-Signal für die Laufwerke liefert. Als Eingabe dient dabei dann nur das Datenbit D0. Ist es gesetzt, werden die Laufwerkmotoren gestartet; ist es null, werden sie gestoppt.

```
    Adresse                          Daten

OUT &X111110x0011111xx,&Xxxxxxxx0 → Stoppe Motor
OUT &X111110x0011111xx,&Xxxxxxxx1 → Starte Motor
            ↑↑↑
          A10 | A7                  D0
              A8
```

Bei diesen Port-Adressen sind die mit x gekennzeichneten Bits beliebig. Die Einser müssen gesetzt sein, damit sich nicht noch andere Geräte angesprochen fühlen. Sie werden aber, wie üblich, nicht überprüft. Die Nullbits dienen zur Adressierung des Flip-Flops und werden Hardware-mäßig ausdecodiert. Insgesamt kann man, aufgrund der 3 x-Bits, acht verschiedene, gültige Port-Adressen benutzen. AMSDOS benutzt normalerweise die Adresse &FA7E, und als Datenbyte nur die Werte &FF (255) und 0.

Beim Z80-Befehl OUT (C),C wird das B-Register auf der oberen Adreßhälfte (A8 bis A15) und das C-Register gleichzeitig auf A0 bis A7 und auf dem Datenbus ausgegeben. Dabei ist für die Adresse die Leitung A0 uninteressant, was Bit 0 des C-Registers entspricht. Gleichzeitig ist nur D0 als Datenelement interessant, was wieder durch Bit 0 des C-Registers bestimmt würde.

Daraus ergibt sich, daß man den Laufwerkmotor in einem Assembler-Programm besonders elegant mit folgenden beiden Werten an- und ausstellen kann:

```
  Motor an:          Motor aus:

LD    BC,#FA7F     LD    BC,#FA7E
OUT   (C),C        OUT   (C),C
```

Die Ein- oder Ausschaltinformation ist dann bereits in die Adresse integriert.

Eine weitere Adresse ist wichtig: &DFxx. Hier ist nur das Adreßbit A13 Null. Das ist keine spezifische Adresse für den Floppy-Controller, sondern allgemein das Signal dafür, daß das Betriebssystem ein neues ROM anwählen will. Das geschieht dabei unabhängig von der ULA: Diese entscheidet nur, *ob* ein ROM eingeblendet wird, aber nicht *welches* es sein soll.

Beim CPC 464 im Grundzustand gibt es da keine Probleme: Hier hat man nur ein ROM, in dem der BASIC-Interpreter untergebracht ist. Sobald man aber den Floppy-Controller hinten ansteckt, und beim CPC 664 und 6128 sowieso, gibt es bereits mindestens 2 ROMs, die im obersten Adreßviertel der CPU residieren.

Diese werden mit einem OUT-Befehl mit der Adresse &DFFF angewählt (aber noch nicht unbedingt eingeblendet, das macht ja die ULA). Das Datenbyte muß dabei die Nummer des gewünschten ROMs enthalten:

```
OUT &DFFF,0        BASIC-ROM
OUT &DFFF,7        AMSDOS-ROM
```

Genaueres darüber folgt aber noch in einem eigenen Kapitel.

PROGRAMMIERUNG DES FDC 765

Der FDC 765 ist ein ausgesprochen intelligenter Floppy-Controller. Das schlägt sich in entsprechend mächtigen Befehlen nieder, mit denen er programmiert werden kann. So ist er beispielsweise in der Lage, nur mit den absolut notwendigen Informationen versehen, eine ganze Spur auf der Diskette zu formatieren. Sektoren zu lesen und zu beschreiben ist mit ihm fast ein Kinderspiel.

Trotz seiner Intelligenz hat er natürlich von der Organisation der Datenspeicherung auf der Diskette keine Ahnung. Der FDC 765 hat mit Inhaltsverzeichnis, Blocks oder Records nichts zu tun. Das wird alles durch die Treibersoftware im AMSDOS-ROM erledigt. Der Floppy-Controller selbst übernimmt nur die Organisation bis zur Sektorebene.

Obwohl er intern über eine Unzahl von Registern verfügt, sind nach außen hin nur zwei verschiedene ansprechbar. Das wird durch einen logischen Kniff erreicht: Ein Register ist ständig ansprechbar und liefert so jederzeit elementare Informationen über den Zustand des FDC. Das ist das sogenannte Haupt-Statusregister. Über die andere Adresse rollt der gesamte Datentransfer. Die Bedeutung der einzelnen Bytes, die man hier hineinschreibt oder hier ausliest,

ist ausschließlich durch ihre zeitliche Abfolge im gesamten Datenstrom festgelegt. Dadurch ist auch jederzeit entweder nur Lesen oder nur Beschreiben dieses "Registers" zulässig.

Ein Befehl des FDC gliedert sich dabei grundsätzlich in drei Phasen:

1. Kommandophase:

Sie wird vom ersten Datenbyte eingeleitet, das in seinen Bits verschlüsselt einen der 15 möglichen Befehle enthält. Danach kommen noch bis zu 8 Parameterbytes, die alle geliefert werden müssen, damit die Reihenfolge stimmt.

2. Ausführungsphase:

Hierin werden beispielsweise beim Sektorschreiben oder -lesen die Datenbytes zwischen Floppy-Controller und CPU übermittelt. Bei einigen Befehlen fehlt dieser Abschnitt.

3. Ergebnisphase:

Bis auf drei Befehle haben alle eine "Resultphase". Hier muß die CPU alle angebotenen Informationen vom FDC abholen, da dieser sonst nicht weiß, wann ein neuer Befehl anfängt. Erst nach dem letzten Byte der Resultphase kann ein neuer Befehl gestartet werden.

Das Haupt-Statusregister und das Datenregister werden beim I/O-Zugriff der CPU durch die Adreßleitung A0 des FDC unterschieden, die direkt an A0 der CPU angeschlossen ist. Da die allgemeine Selektion des FDC wie beim Motor-ON-Flip-Flop über die Adreßbits A10 und A7 erfolgt, ergeben sich folgende Adressen für diese beiden Register:

&X111110x1011111x0 → Haupt-Statusregister
&X111110x1011111x1 → Datenregister

Die x-Bits in der Adresse sind wieder beliebig. Von den jeweils vier möglichen gültigen Adreßkombinationen werden von AMSDOS folgende benutzt:

&FB7E → Haupt-Statusregister
&FB7F → Datenregister

Hierbei wird von der Hardware außer den verpflichtenden Null-Pegeln an A10 und A7 auch ausnahmsweise einmal ein Eins-Pegel getestet: A8 muß gesetzt sein und dient zur Unterscheidung von der Adresse des Motor-Flip-Flops.

Das Haupt-Statusregister – INP(&FB7E)

Dieses Register kann ständig gelesen (was bei den Datenregistern nicht der Fall ist), dafür aber nicht beschrieben werden. Hier kann man jederzeit die wichtigsten Informationen über den Zustand des FDC 765 erhalten:

Bit 7 – RQM – ReQuest for Master

0	kein Zugriff auf den FDC möglich
1	Datenübertragung oder Befehl möglich (je nach Bit 4, 5 und 6)

Nur wenn dieses Bit gesetzt ist, ist der FDC bereit, ein Datenwort zu empfangen oder auszugeben. Da der Floppy-Controller im Schneider CPC mittels Polling betrieben wird, muß die CPU bei einer Datenübertragung beispielsweise ständig dieses Bit testen, um zu erfahren, ob das nächste Byte für die Übertragung zur Floppy benötigt wird.

Bit 6 – DIO – Data Input/Output

0	Datenrichtung ist: CPU → FDC
1	Datenrichtung ist: FDC → CPU

Bit 6 liefert dazu noch eine wichtige Zusatzinformation: die Datenrichtung. Dieses Bit signalisiert, ob der FDC ein Byte für die CPU hat (Lesedaten von der Floppy oder Statusinformationen) oder ob er noch eins benötigt (Schreibdaten oder Befehlsbytes).

Bit 5 – EXM – Execution Mode

0	Auswertungsphase
1	Ausführungsphase

Im DMA-Betrieb wird dieses Bit nicht benutzt und ist immer 0. Ansonsten kann die CPU hieran unterscheiden, ob die vom FDC gelieferten Bytes Daten von einem Sektor sind, der gerade gelesen wird, oder ob sie bereits aus der Resultphase stammen.

Bit 4 – FCB – Floppy Controller Busy

0	der FDC ist frei; ein neuer Befehl kann gestartet werden.
1	besetzt; der FDC arbeitet gerade einen Befehl ab.

Dieses Flag wird gesetzt, sobald der FDC die Bearbeitung eines Befehls aufgenommen hat, oder anders ausgedrückt: sobald man das erste Byte eines neuen Befehls in das Datenregister geschrieben hat. Es wird erst mit dem Ende der Auswertungsphase wieder zurückgesetzt. Damit läßt sich beispielsweise in Multi-Tasking-Systemen der Diskettenzugriff zwischen den parallel laufenden Programmen steuern.

Bits 0-3 – FDB – Floppy Drive Busy

0	normal
1	SEEK oder RECALIBRATE läuft auf dem entsprechenden Drive

Diese Bits sind den 4 möglichen Laufwerken zugeordnet und werden gesetzt, sobald man das Kommando SEEK (Spur suchen) oder RECALIBRATE (Spur 0 suchen) startet. Bis zum Ende eines solchen Befehls sind nur noch diese Befehle auf anderen Laufwerken möglich.

Das Statusregister 0

Bits 7 und 6 – Interrupt Code

00	Kommando erfolgreich beendet
01	Kommando abgebrochen wegen Fehler
10	ungültiges Kommando
11	Kommando abgebrochen, weil Drive "not ready" wurde

Bei fast allen Sektor-Schreib- oder Lesebefehlen liefert der FDC in der Ergebnisphase in diesem Register die Meldung "Kommando abgebrochen wegen Fehler". Ursache ist die schon beschriebene Beschaltung des TC-Anschlusses (Terminal Count). In diesem Fall muß man untersuchen, ob es sich bei dem Fehler um "End of Track" (Bit 7 von Statusregister 1) handelt. Dann liegt im Schneider CPC kein Fehler vor!

Nach einem SEEK oder RECALIBRATE erzeugt der FDC einen Interrupt, worauf die CPU mit dem Befehl &08 "Sense Statusregister 0" dieses Register hier abfragen muß, um das Kommando zu beenden.

Da der INT-Ausgang des FDC aber gar nicht angeschlossen ist, muß die CPU ständig das Statusregister 0 abfragen, bis der FDC meldet, daß das Kommando beendet ist. Ist es das noch nicht, meldet er immer den Fehler 10: ungültiges Kommando.

Bit 5 = 1 – Seek-Ende auf einem Drive

Nach dem Befehl SEEK oder RECALIBRATE wird dieses Bit gesetzt, wenn dieser Befehl abgeschlossen wurde.

Bit 4 = 1 – Fehler in der Floppy

Dieses Flag wird gesetzt, wenn die Floppy das Fehler-Flip-Flop setzt oder wenn nach einem RECALIBRATE nach 77 Schrittimpulsen immer noch kein Track0-Signal gemeldet wird. Das kann bei 80-Spur-Laufwerken auch ganz normal passieren, wenn der Schreib-/Lesekopf auf den innersten Spuren steht. Beide Gründe scheiden aber bei den 3-Zoll-Floppies aus. Diese haben ja nur 40 Spuren, und durch die Beschaltung des Anschlusses FLT/Track0 des FDC (s.o.) kann auch der Grund nicht beim Fehler-Flip-Flop liegen. Ist dieses Bit im AMSDOS-Controller gesetzt, bewegt sich der Schrittmotor in der Floppy nicht! Das dürfte eigentlich nur beim CPC 664 oder 6128 möglich sein, wenn man die 12-Volt-Versorgung herauszieht.

Bit 3 = 1 – Laufwerk ist nicht bereit

Startet man einen Befehl auf einem Laufwerk, in dem beispielsweise keine Diskette eingelegt ist, so meldet es "not ready" am entsprechenden Eingang des Controllers. Das geschieht normalerweise ebenfalls, wenn man bei den einseitigen Laufwerken versucht, auf die zweite Seite zuzugreifen; nicht jedoch bei den Schneider-Floppies.

Bit 2 – Head

0 – Seite 1 der Diskette
1 – Seite 2 (nur Doppelkopf-Laufwerke)

Dieses Bit gibt den momentan angewählten Schreib-/Lesekopf an. Die Schneider-Floppies sind aber alle einseitig. Dadurch ist hier nur ein Zugriff auf Kopf 1 (Bit 2 = 0) möglich.

Bit 1 und 0 – Unit Select

x0 – Drive A
x1 – Drive B

Diese Bits geben das momentan angewählte Laufwerk an. Sie entsprechen den US-Ausgängen des FDC. Im Schneider CPC wird nur Bit 0 benutzt.

Das Statusregister 1

Die Statusregister 1 und 2 spezifizieren die Fehlermeldungen aus den Bits 6 und 7 des Statusregisters 0.

Bit 7 = 1 – End of track error

Dieses Flag zeigt an, daß der Floppy-Controller bei einem Multi-Sektor-Read oder -Write auf einen Sektor zugreifen will, der den programmierten "letzten Sektor des Tracks" (siehe Befehlsbeschreibungen) überschreitet. Der FDC führt automatisch solche Vielfach-Lese- und -Schreibzyklen durch, wenn er nicht durch einen Impuls am TC-Anschluß gestoppt wird. Durch die Beschaltung dieses Anschlusses im Schneider CPC wird dieser Fehler nach jedem Lese- oder Schreibzugriff gemeldet!

Bit 6 – unbenutzt (immer 0)

Bit 5 = 1 – CRC-Fehler im Daten- oder ID-Feld

Der FDC 765 erzeugt beim Schreiben eines Sektors mehrere Prüfsummen nach dem CRC-Verfahren, die mit auf der Diskette aufgezeichnet werden. Stimmt beim Lesen diese Prüfsumme nicht, dann wird dieses Flag gesetzt.

Bit 4 = 1 – Pufferüberlauf

Beim Datentransfer vom oder zum Sektor, der gerade geschrieben oder gelesen wird, muß der Prozessor die Bytes in extrem kurzen Zeitabständen übertragen. Kommt er einmal nicht nach, so gerät der Datenfluß zwischen Controller und Floppy ins Stocken. Da der Floppy-Controller die Diskette aber nicht blitzartig anhalten kann, gehen entweder Daten verloren (beim Lesen) oder können nicht geschrieben werden. In diesem Fall wird dieses Flag gesetzt. Das sollte aber höchstens während der Entwicklungsphase eines neuen Rechners vorkommen.

Bit 3 – nicht benutzt (immer 0)

Bit 2 = 1 – Sektor nicht auffindbar

Dieser Fehler tritt auf, wenn der angegebene Sektor bei einem Schreib- oder Lesebefehl auf der Spur nicht gefunden werden kann.

Bit 1 = 1 – Diskette ist schreibgeschützt

Bei einem Schreibversuch auf die Diskette wird dieses Flag gesetzt, wenn dort die Schreibschutzmarke gesetzt ist. Dann werden selbstverständlich keine Daten aufgezeichnet. Mit dem FDC 765 ist es nicht möglich, auf eine schreibgeschützte Diskette zu schreiben, auch wenn die Laufwerkselektronik dies zuließe. Dort gibt es aber meist auch solche Präventiv-Maßnahmen. Als Schreibversuche zählen die Sektor-Schreibbefehle und das Formatierkommando.

Bit 0 = 1 – ID oder Data Address Mark fehlt

Hiermit wird angezeigt, daß der FDC mit der Formatierung der Spuren nicht zurechtkommt. Findet er die Adreßmarken nicht, so wird dieses Bit gesetzt. Möglicherweise ist die Spur gar nicht formatiert.

Das Statusregister 2

Bit 7 – unbenutzt (immer 0)

Bit 6 = 1 – "gelöschter" Sektor gefunden
Bit 5 = 1 – Checksummenfehler im Datenteil eines Sektors
Bit 4 = 1 – die logische Spurnummer aus der Sektor-ID stimmt nicht
Bit 3 = 1 – Vergleich von Sektor- und Prozessordaten lieferte Gleichheit
Bit 2 = 1 – Testbedingung im Scan-Kommando nicht erfüllt
Bit 1 = 1 – Track enthält fehlerhafte Stellen. Nicht beschreiben!
Bit 0 = 1 – Die Markierung für den Datenbereich war nicht auffindbar

Das Statusregister 3

Dieses Register spiegelt den aktuellen Zustand der wichtigsten Signale vom angewählten Laufwerk wieder. Dieses Register kann nur mit einem speziell dafür eingerichteten Befehl (&04) gelesen werden.

Bit 7 = 1 – Fehler-Flip-Flop gesetzt

Wegen der Beschaltung dieses Eingangs des FDC wird dieses Bit im Schneider CPC nie gesetzt sein.

Bit 6 = 1 – eingelegte Diskette ist schreibgeschützt

Bit 5 = 1 – Laufwerk meldet "ready"

Normalerweise müßten die einseitigen Schneider-Laufwerke sich weigern, "ready" zu melden, wenn man auf Seite 2 zugreifen will. Das tun sie aber nicht. Sehr wahrscheinlich ist das Signal "side select" dort gar nicht angeschlossen.

Bit 4 = 1 – Schreib-/Lesekopf steht auf Spur Null (Track 0)

Bit 3 = 1 – Doppelkopf-Laufwerk

Dieses Signal wird über denselben Pin des FDC-ICs eingelesen wie das Schreibschutzsignal. Es gehört zu der Gruppe von vier Pins, deren Bedeutung vom Ausgang RW/SEEK umgeschaltet werden kann. Die Schneider-Floppies ignorieren aber ganz offensichtlich den Umschaltbefehl beim Doppelausgang FLT/TRK0 und bei WRPRT/2SIDE. Für die Bedeutung "Fault" wurde durch externe Beschaltung vorgesorgt, daß eine Fehlermeldung nicht bis zum FDC 765 gelangt, weil dieser sonst ständig seine Arbeit abbrechen würde. Die Funktion 2SIDE liefert jedoch immer den Zustand von WRTPRT (schreibgeschützt). Dieses Flag ist daher immer wie Bit 6 gesetzt und kann nicht benutzt werden, um Doppelkopf-Laufwerke zu erkennen.

Die folgenden Bits spiegeln noch einmal die im Befehl angegebenen Werte für Kopfseite und Unit Select 0 und 1 wieder:

Bit 2 = aktuell angewählte Diskettenseite (Head Address)
Bit 1 = aktueller Zustand von US1
Bit 0 = aktueller Zustand von US0

DIE BEFEHLE DES FDC 765

Der FDC 765 kennt insgesamt 15 verschiedene Befehle, die zum Teil äußerst komplexe Funktionen erfüllen. Diese sind im folgenden nach ihrem Befehlscode sortiert aufgeführt. Allgemein muß man bei allen Befehlen drei Phasen unterscheiden:

1. Befehlsphase
2. Ausführungsphase
3. Ergebnisphase

Bei einigen Befehlen fehlen die Phasen 2 und/oder 3. In der Befehlsphase können Daten nur geschrieben und in der Ergebnisphase nur gelesen werden. Die Datenrichtung in der Ausführungsphase hängt vom gewählten Befehl ab.

Die verwendeten Abkürzungen bedeuten:

t – Multi Track Bit:

Zweiseitige Disketten werden oft so organisiert, daß eine Spur auf der Rückseite die logische Fortsetzung der Spur auf der Vorderseite ist. Die Befehle des FDC, die mehrere Sektoren bearbeiten können, unterstützen das, wenn dieses Bit gesetzt ist. Unter AMSDOS ist das natürlich nicht anwendbar und daher immer auf 0 gesetzt.

m – MFM-Modus:

Allen Befehlen, die irgendwie auf die Diskette zugreifen, muß mitgeteilt werden, ob sie im MF-Modus (einfache Dichte) oder MFM-Modus (doppelte Dichte) arbeiten sollen. Unter AMSDOS ist dieses Bit immer gesetzt, weil die Disketten alle ausschließlich mit doppelter Schreibdichte formatiert werden.

s – Skip deleted DAM:

Alle Sektoren können vom FDC als "gelöscht" oder "nicht gelöscht" geschrieben werden. Das hat jedoch nichts mit gelöschten Dateien zu tun, sondern bezieht sich nur auf eine Markierung in der Data Address Mark (DAM) jedes Sektors. Ist dieses Bit gesetzt, betrachtet der FDC alle gelöschten Sektoren als nicht existent. Unter AMSDOS ist das nicht einsetzbar und dieses Flag deshalb immer 0.

drv – Head and Drive:

Fast alle Befehle verlangen eine Information darüber, welches Laufwerk und welche Seite angesprochen werden soll. Die drei Bits drv spiegeln dabei direkt die Ausgangssignale HEAD, US1 und US0 wieder. Von AMSDOS werden nur einseitige Laufwerke unterstützt, deshalb ist das erste Bit immer 0. Außerdem ist US1 nicht angeschlossen, weshalb das mittlere Bit normalerweise auch nicht gesetzt wird. Nur US0, das dem rechten der drei Bits entspricht, wird benutzt, um zwischen Drive A und Drive B zu unterscheiden:

drv = &X000 – Drive A
drv = &X001 – Drive B

x:

Bits, deren Wert nicht ausgewertet wird.

OUT Standard-Datenblock:

Die meisten Befehle des FDC 765 erfordern u. a. sieben Bytes, die immer dieselbe Bedeutung haben:

```
OUT &FB7F,Spurnummer          ; Sektor-ID-Information
OUT &FB7F,Kopfnummer          ; dito
OUT &FB7F,Sektornummer        ; dito
OUT &FB7F,Sektorgroesse       ; dito
OUT &FB7F,logische letzte Sektornummer der Spur
OUT &FB7F,Lücke zwischen Sektor-ID und Daten
OUT &FB7F,Sektorlaenge wenn Sektorgroesse = 0
```

INP Standard-Datenblock:

Ebenso ist es in der Ergebnisphase. Hier gibt der FDC fast immer dieselben sieben Bytes aus:

```
INP &FB7F,Statusregister 0
INP &FB7F,Statusregister 1
INP &FB7F,Statusregister 2
INP &FB7F,Spurnummer          ; Sektor-ID-Informationen
INP &FB7F,Kopfnummer          ; dito
INP &FB7F,Sektornummer        ; dito
INP &FB7F,Sektorgroesse       ; dito
```

&X0ms00010 – ganze Spur lesen (READ TRACK)

Befehlsphase:

```
OUT &FB7F,&X0ms00010          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT Standard-Datenblock
```

Ausführungsphase:

Datentransfer FDC → CPU

Ergebnisphase:

INP Standard-Datenblock

&X00000011 – Laufwerksdaten festlegen (SPECIFY)

Befehlsphase:

```
OUT &FB7F,&X00000011          ; Befehl
OUT &FB7F,&Xssssuuuu          ; Steprate & Head unload time
```

```
OUT &FB7F,&Xllllllld          ; Head load time & DMA-Flag
```

Dieser Befehl sollte nach Möglichkeit der erste überhaupt sein, den man zum Floppy-Controller schickt. Hiermit werden drei verschiedene Timing-Parameter festgelegt und der FDC auf DMA-Betrieb eingestellt (oder auch nicht). Die einzelnen Bits bedeuten:

ssss	Steprate für den Schrittmotor. Die Wartezeit zwischen den einzelnen Impulsen beträgt (16-ssss)*2 msek.
uuuu	Head unload time. Zeit, die der FDC automatisch nach jedem Schreib- oder Lesevorgang wartet. Wird nur bei Laufwerken benötigt, die ihren Kopf tatsächlich von der Floppy Disk abheben können. Die Wartezeit beträgt uuuu*32 msek.
lllllll	Head load time. Ebenfalls nur bei Laufwerken interessant, die den Schreib-/Lesekopf tatsächlich abheben können. Diese Zeit wartet der FDC vor jedem Diskettenzugriff. Die Wartezeit beträgt (lllllll+1)*4 msek.
d	DMA-Flag. Ist dieses Bit gesetzt, benutzt der FDC seine Steuersignale für einen DMA-Controller. Die CPU kann dann während der Datenübertragung weiterrechnen.

Die angegebenen Zeiten beziehen sich auf einen Eingangstakt von 4 MHz an Pin CLK des Floppy-Controllers.

&X00000100 – Statusregister 3 abfragen (SENSE DRIVE STATE)

Befehlsphase:

```
OUT &FB7F,&X00000100          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
```

Ergebnisphase:

```
INP &FB7F,Statusregister 3
```

Dieser Befehl stellt die einzige Möglichkeit dar, das Statusregister 3 abzufragen. Hier liefert der FDC dann die wichtigsten Zustandssignale vom angewählten Laufwerk.

&Xtm000101 - Sektor(en) schreiben (WRITE SECTOR)

Befehlsphase:

```
OUT &FB7F,&Xtm000101          ; Befehl
```

```
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Datenübertragung CPU → FDC → Floppy

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

Der FDC sucht den im Standard-Datenblock angegebenen Sektor und fordert dann so lange Bytes von der CPU bzw. dem DMA-Controller an, bis der Sektor mit Daten vollgeschrieben ist. Erfolgt dann kein Impuls am TC-Eingang, macht der FDC automatisch mit dem nächsten Sektor weiter.

&Xtms00110 – Sektor(en) lesen (READ SECTOR)

Befehlsphase:

```
OUT &FB7F,&Xtms00110          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Datenübertragung Floppy → FDC → CPU

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

In der Ausführungsphase muß die CPU oder der DMA-Controller die Datenbytes aus dem Sektor abholen. Auch hier macht der FDC ein "Multi Sektor Read", wenn er nicht durch einen TC-Impuls gestoppt wird.

&X00000111 - Spur 0 suchen (RECALIBRATE)

Befehlsphase:

```
OUT &FB7F,&X00000111          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
```

Ausführungsphase:

Der FDC erzeugt für das ausgewählte Laufwerk so lange Schrittimpulse, bis dort der Spur-0-Sensor aktiv wird, maximal jedoch 77 Stück. Danach erzeugt der FDC einen Interrupt-Impuls, und die CPU muß mit dem Befehl &08 das Interrupt-Statusregister lesen, um den Befehl zu beenden. Vorher können keine neuen Befehle gestartet werden, außer SEEK und RECALIBRATE auf anderen Laufwerken.

&X00001000 – Statusregister 0 abfragen (SENSE INTERRUPT STATE)

Befehlsphase:

```
OUT &FB7F,&X00001000          ; Befehl
```

Ergebnisphase:

```
INP &FB7F,Status 0
INP &FB7F,Spurnummer
```

Wenn der FDC 765 nicht im DMA-Betrieb eingesetzt wird, erzeugt er zu vier verschiedenen Gelegenheiten Interrupts:

1. Für jedes Byte während der Ausführungsphase
2. Zu Beginn der Ergebnisphase
3. Mit dem Ende eines SEEK oder RECALIBRATE
4. Wenn sich das Ready-Signal eines Laufwerks ändert

Das Programm muß nun feststellen, wieso ein Interrupt ausgelöst wurde. Fall 1 und 2 treten während der Bearbeitung eines Befehls auf und können leicht durch das Haupt-Statusregister erkannt werden. Außerdem sollten die Programme so aufgebaut sein, daß hier bei einem Interrupt schon vorher klar ist, was da vom FDC auf den Prozessor zukommt.

Fall 3 und 4 sind schwieriger zu erkennen. SEEK und RECALIBRATE werden nur durch den Befehl gestartet. Zwar kann, solange der Controller noch mit einem solchen Kommando beschäftigt ist, mit keinem anderen begonnen werden (außer denselben auf anderen Laufwerken), trotzdem ist der Befehl für die CPU abgearbeitet. Diesen Befehlen fehlt nämlich die Ergebnisphase!

Und der "ready"-Status eines Laufwerks kann sich jederzeit ändern, wenn der Benutzer die Diskette aus dem Laufwerk nimmt. Das ist zu unterscheiden von

einem "not ready"-Werden des Laufwerks im Verlauf eines Befehls. Das erkennt der Prozessor in der Ergebnisphase.

Erzeugt der Floppy-Controller also ein Interrupt-Signal, muß die CPU das Statusregister 0 abfragen. Dieses wird deshalb auch Interrupt-Statusregister genannt. Im AMSDOS-Controller des Schneider CPC ist der Interrupt-Ausgang des FDC jedoch nicht angeschlossen.

&Xtm001001 – gelöschte Sektoren schreiben (WRITE DELETED SECTOR)

Befehlsphase:

```
OUT &FB7F,&Xtm001001          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Datentransfer CPU → FDC → Floppy

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

Dieser Befehl entspricht vollkommen dem Kommando "Sektor Schreiben". Einziger Unterschied ist, daß in der Data Address Mark (DAM) eine andere Kennung eingetragen wird.

&X0m001010 – Sektor-ID lesen (READ SECTOR ID)

Befehlsphase:

```
OUT &FB7F,&X0m001010          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
```

Ausführungsphase:

FDC liest die erste Sektor-ID, auf die er auf dem angewählten Laufwerk trifft.

Ergebnisphase

```
INP &FB7F,Standard-Datenblock
```

Jeder Sektor hat eine vier Bytes lange Kennung, seine "ID", in der Spur-, Kopf- und Sektornummer und die Sektorgröße eingetragen sind. Diese Sektor-ID ist Bestandteil der Standard-Datenblocks der Befehls- und Ergebnisphase.

&Xtms01100 – gelöschte Sektoren lesen (READ DELETED SECTOR)

Befehlsphase:

```
OUT &FB7F,&Xtms01100          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Datentransfer Floppy → FDC → CPU

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

Dieser Befehl entspricht vollkommen dem Kommando "Sektor Lesen". Einziger Unterschied ist, daß in der Data Address Mark (DAM) eine andere Kennung eingetragen sein muß.

&X0m001101 – eine Spur formatieren (FORMAT TRACK)

Befehlsphase:

```
OUT &FB7F,&X0m001101          ; Befehl
OUT &FB7F,&X0m001drv          ; Drive
OUT &FB7F,Sektorgrösse
OUT &FB7F,Anzahl Sektoren
OUT &FB7F,Lücke zw. ID und Daten
OUT &FB7F,Füllbyte für den Datenbereich
```

Ausführungsphase:

Der FDC formatiert die Spur mit allem, was dazugehört und fordert für jeden Sektor von der CPU (DMA) 4 Bytes für die Sektor-ID an.

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

&X00001111 – Spur suchen (SEEK)

Befehlsphase:

```
OUT &FB7F,&X00001111          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Spurnummer
```

Ausführungsphase:

Der FDC vergleicht die gewünschte (physikalische) Spurnummer mit der aktuellen, die er für jedes Laufwerk in einem Register speichert.

Der FDC kann ja nur die Spur 0 erkennen und ist ansonsten darauf angewiesen, selbst mitzuzählen. Dann erzeugt er so viele Schrittimpulse, wie notwendig sind, um die aktuelle Spurnummer mit der gewünschten in Deckung zu bringen. In dieser Zeit kann der FDC keine weiteren Befehle entgegennehmen, außer SEEK und RECALIBRATE auf anderen Laufwerken.

Ist die gewünschte Spur erreicht, erzeugt der FDC einen Interrupt-Impuls, worauf die CPU das Statusregister 0 lesen muß, um den Befehl endgültig abzuschließen.

&Xtms10001 – Sektor(en) testen (SCAN EQUAL)

Befehlsphase:

```
OUT &FB7F,&Xtms10001          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Der FDC liest den angegebenen Sektor und vergleicht die einlaufenden Datenbytes mit denen, die die CPU (oder DMA) liefert.

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

Im Prinzip läuft dieses Kommando genauso ab, wie beim Sektorlesen oder -schreiben, nur werden hierbei die Daten aus Speicher und Floppy verglichen. Erfolg oder Mißerfolg kann man dann nachher im Statusregister 2 prüfen. SCAN EQUAL testet nur auf exakte Datengleichheit (Verify).

&Xtms11001 – Sektor(en) testen (SCAN LOW OR EQUAL)

Befehlsphase:

```
OUT &FB7F,&Xtms11001          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Hier ist es wie bei SCAN EQUAL, nur dürfen die Bytes aus dem Sektor auch *kleiner* als ihr jeweiliges Pendant aus dem Speicher des Computers sein.

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

&Xtms11101 – Sektor(en) testen (SCAN HIGH OR EQUAL)

Befehlsphase:

```
OUT &FB7F,&Xtms11101          ; Befehl
OUT &FB7F,&Xxxxxxdrv          ; Drive
OUT &FB7F,Standard-Datenblock
```

Ausführungsphase:

Hier ist es ebenfalls wie bei SCAN EQUAL, nur dürfen die Datenbytes von der Diskette auch *größer* als ihr jeweiliger Vergleichspartner sein.

Ergebnisphase:

```
INP &FB7F,Standard-Datenblock
```

ORGANISATION EINER SPUR DURCH DEN FDC 765

Der Unterschied zwischen einer nagelneuen und einer formatierten Diskette ist ganz beträchtlich: Schmücken sich einige Hersteller mit der Bezeichnung "250 kByte Datenkapazität pro Diskettenseite" so bleiben nach dem Formatieren davon nur noch genau 184320 Bytes übrig (wovon dann noch Inhaltsverzeichnis und Systemspuren abgehen). Immerhin ein glattes Viertel ist verschwunden.

Tatsächlich lassen sich bei den 3-Zoll-Laufwerken auf einer Diskettenseite maximal 250000 Bytes unterbringen. Die sind aber nicht alle für Daten verfügbar. Die Formatierung selbst beansprucht eben auch Platz, und, wie in den folgenden Tabellen zu sehen ist, ist der Verwaltungs-Overhead ganz beachtlich.

EINE SPUR AUF DER DISKETTE

1. Indeximpuls markiert Spuranfang

2. Gap 4A – Preamble – 80 Bytes mit dem Wert &4E

   Diese Lücke (Gap) soll den Unterschied zwischen verschieden justierten Lichtschranken bei anderen Laufwerken ausgleichen.

3. Spur-Header:

   3.1. Synchronisation – 12 Bytes mit dem Wert &00

   3.2. Index Address Mark – IAM – 3 Bytes

   3.3. 1 Byte &FC

4. Gap 1 – Spacing – 50 Bytes mit dem Wert &4E

   Diese Lücke soll dem Floppy-Controller Zeit genug lassen, die IAM zu verarbeiten.

5. Es folgen die Sektoren (siehe unten)

6. Gap 4B – Postamble – Bytes &4E bis zum Spurende

   Dieser Nachspann dient dazu, unterschiedliche Umdrehungsgeschwindigkeiten verschiedener Laufwerke auszugleichen. Der FDC schreibt nach dem Formatieren des letzten Sektors so lange Bytes mit dem Wert &4E, bis ein neuer Impuls von der Indexloch-Lichtschranke eintrifft.

EIN SEKTOR IN EINER SPUR

1. Identifikation – Sektor-ID

   1.1. Synchronisation – 12 Bytes &00

1.2. Identification Address Mark – IDAM – 3 Bytes

1.3. 1 Byte &FE

1.4. Identification Field – Sektor-ID (siehe unten)

1.5. CRC Checksumme – 2 Bytes

2. Gap 2 – 22 Bytes &4E

   Diese Lücke läßt dem FDC Zeit, die Sektor-ID zu bearbeiten.

3. Datenteil

   3.1. Synchronisation – 12 Bytes &00

   3.2. Data Address Mark – DAM – 3 Bytes

   3.3. 1 Byte &FB

   3.4. Datenfeld

   Es folgen so viele Bytes, wie beim Formatieren der Spur für jeden Sektor festgelegt wurden.

   3.5. CRC Prüfsumme – 2 Bytes

4. Gap 3 – 54 Bytes &4E

   Diese Lücke muß die Toleranzen in der Rotationsgeschwindigkeit zwischen verschiedenen Laufwerken ausgleichen. Da der Datenteil eines Sektors (und nur dieser) beim Sektorschreiben neu beschrieben wird, können schneller drehende Drives einen längeren Kreisausschnitt beschreiben, als vorher formatiert wurde. Dann steht das neue Datenfeld hintenüber. Wäre Gap 3 nicht, so würde die Synchronisation für den nächsten Sektor überschrieben.

DAS IDENTIFIKATIONSFELD IN JEDEM SEKTOR

Diese sogenannte Sektor-ID ist Bestandteil jedes Sektors und muß bei allen FDC-Befehlen, die mit Sektoren arbeiten, angegeben werden. Sie ist sowohl Bestandteil des Standard-Datenblocks der Befehlsphase als auch der Ergebnisphase.

Beim Formatieren einer Spur kann zu jedem Sektor das ID-Feld neu bestimmt werden. Da zum Lesen eines Sektors wieder die genaue Kenntnis des gesamten ID-Feldes nötig ist, kann man das mitunter benutzen, um Geheimsektoren in einer Spur unterzubringen, die die Diskette vor Raubkopierern schützen. Sehr wirkungsvoll ist das leider auch nicht, da man mit dem Befehl "Sektor-ID lesen" ziemlich sicher alle Sektoren auf einer Spur finden kann.

Das ID-Feld ist genau vier Bytes lang. Diese Bytes enthalten folgende Angaben:

1. Track – Cylinder – Spurnummer

   Jede Spur kann unabhängig von ihrer tatsächlichen physikalischen Lage eine "logische" Spurnummer erhalten. So ist es beispielsweise möglich, die Datenspuren einer Diskette erst ab der physikalischen Spur 2 mit 0 beginnend durchzunumerieren, weil die äußeren beiden Systemspuren sind. Doppelseitige Disketten können eventuell so formatiert werden, daß auf der Vorderseite nur gerade und auf der Rückseite nur ungerade Spurnummern sind. In jedem Fall hängt die logische Sektornumerierung ausschließlich vom DOS (Disketten-Verwaltungsprogramm) ab. AMSDOS numeriert die logischen Spurnummern in allen Diskettenformaten entsprechend ihrer physikalischen Lage ab 0 durch.

2. Diskettenseite

   Wieder unabhängig von der tatsächlichen Diskettenseite kann man hier eine logische Seitennummer eintragen. Meist wird man sich aber an das Head-Select-Bit in den FDC-Befehlen halten: Seite 1 erhält die Seitennummer 0, Seite 2 die Seitennummer 1. Unter AMSDOS werden alle Sektoren mit der Seitennummer 0 formatiert.

3. Sektornummer

   Oft ist es sehr nützlich, die Sektoren unabhängig von ihrer tatsächlichen Reihenfolge in der Spur durchnumerieren zu können. Mitunter kann man den Diskettenzugriff eines Programms erheblich beschleunigen, indem man die Sektoren nicht nacheinander, sondern mit einem bestimmten Versatz aufbringt:

   1-3-5-7-9-2-4-6-8 oder ähnlich.

   Unter AMSDOS werden die verschiedenen Formate durch unterschiedliche Sektornummern-Offsets markiert:

   IBM-Sektoren von 1 bis 8

CP/M-Sektoren von &41 bis &49
Daten-Sektoren von &C1 bis &C9.

4. Sektorgröße

Im allgemeinen muß hier die reale Sektorgröße in der codierten Form angegeben werden, wie sie auch beim Formatieren benutzt wird. Gibt man jedoch eine Null an, so wird die Größe durch das letzte Byte des Standard-Datenblocks der Befehlsphase bestimmt.

DIE BESONDERHEITEN DES FDC 765 IM SCHNEIDER CPC

Wie CRTC und CPU ist auch der Floppy-Controller FDC 765 im Schneider CPC teilweise recht ungewöhnlich eingesetzt.

FAULT

Durch die Beschaltung der Doppelfunktionsanschlüsse FRES/STEP und FAULT/TRK0 kann nie ein Fehlersignal von der Floppy zum Controller gelangen. Verfügt die Floppy aber trotzdem über ein Fehler-Flip-Flop, wird dieses bei jedem Schreib-/Lesezugriff des FDC zurückgesetzt.

Terminal Count

Der Eingang TC (Terminal Count) ist mit RESET zusammengeschaltet. Dadurch ist er vom laufenden Programm nicht ansprechbar und kann prinzipiell als unbeschaltet betrachtet werden.

Dieser Eingang wird jedoch benötigt, um Sektor-Schreib- und Lesezugriffe des FDC abzubrechen. Ohne einen TC-Impuls macht der FDC automatisch mit dem nächsten Sektor weiter und führt sogenannte Multi-Sektor-Reads oder -Writes aus.

Für AMSDOS wurde das Problem so gelöst, daß die in jedem Befehl zu programmierende höchste Sektornummer der Spur dem zu lesenden Sektor gleichgesetzt wird. Der Controller liest den angegebenen Sektor, erhält keinen TC-Impuls und will mit dem darauffolgenden Sektor weitermachen. Dabei stellt er aber fest, daß er den programmierten "letzten Sektor" überschreitet und bricht den Befehl ab.

In der Result-Phase (Ergebnisphase) meldet der FDC dann im Statusregister 0 den Fehler "Befehl abgebrochen". Die Bits 7 und 6 enthalten den Wert &X01.

Im Statusregister 1 wird der Fehler näher spezifiziert: Bit 7 ist gesetzt (End of Track) womit gerade der beschriebene Fehler angezeigt wird.

Dieser Fehler tritt nur beim Befehl "Sektor-ID lesen" nicht auf. Dann liest der FDC nur die nächste erreichbare Sektor-ID auf der Diskette und stoppt auch ohne TC-Impuls. Ausgenommen sind natürlich auch solche Befehle wie "Recalibrate", "Statusregister 0 lesen" o.ä., die nicht auf die Diskette zugreifen.

INT

Der Interrupt-Ausgang ist tatsächlich nirgendwo angeschlossen. Der FDC erzeugt aber bei 4 verschiedenen Gelegenheiten einen Interrupt:

1. Für jedes zu übertragende Byte in der Ausführungsphase
2. Zu Beginn der Ergebnisphase
3. Sobald ein SEEK oder RECALIBRATE beendet wurde
4. Wenn sich das READY-Signal eines Laufwerks ändert

Er erwartet in jedem Fall, daß die CPU darauf reagiert! Nun ist der Interrupt-Ausgang aber nicht angeschlossen. Mithin ist diese Forderung nicht so ohne weiteres zu erfüllen.

Fall 1 und 2 sind leicht zu behandeln. Die CPU hat eine Datenübertragung begonnen und fragt nun ständig im Haupt-Statusregister nach, ob ein Byte übergeben werden muß (erkennbar an Bit 7 – Request for Master). Außerdem kann sie an Bit 5 (Execution Mode) unterscheiden, ob ein Byte noch aus dem zu lesenden Sektor oder bereits aus der Result-Phase stammt.

Fall 3 kann entweder so behandelt werden, daß die CPU so lange wartet, bis sie durch Lesen des Statusregisters 0 (Bit 5 – SEEK END) erkennt, daß der Befehl abgeschlossen ist, oder man setzt ein Flag und fügt diese Warteschleife erst vor dem nächsten Lese- oder Schreibzugriff ein.

AMSDOS speichert in einer System-Variablen die aktuelle Spurnummer (zusätzlich zum FDC und zusätzlich zum Floppy-Laufwerk) und ruft nach jedem SEEK-Kommando eine genau berechnete Warteschleife auf, um danach das Interrupt-Statusregister zu lesen.

Fall 4 ist am schwierigsten: Das READY-Signal der Floppy kann sich praktisch jederzeit ändern; entweder, wenn der Anwender eine Diskette einlegt oder herausnimmt, oder auch, wenn AMSDOS den Laufwerkmotor startet oder stoppt.

Am sinnvollsten ist hierbei, vor jedem Zugriff auf den FDC das Statusregister 0 zu lesen. Liegt nichts vor, quittiert das der FDC mit einem "illegal command", die Bits 7 und 6 sind in der Kombination &X10 gesetzt.

Sonst ist entweder Bit 5 gesetzt (Seek End), oder man kann an Bit 3 erkennen, ob sich der Ready-Status des Laufwerks geändert hat.

Da es zu jedem Laufwerk ein Statusregister gibt, ist es sinnvoll, das Register erneut abzufragen, wenn man etwas anderes als "illegal command" erhielt.

US1

Die beiden möglichen Laufwerke am AMSDOS-Controller werden nur über die Leitung US0 unterschieden. US1 ist nicht angeschlossen.

Das führt dazu, daß beispielsweise eine Änderung des Ready-Signals an Laufwerk A (US0 = 0 und US1 = 0) für den FDC so aussieht, als würde sich der Ready-Zustand auch bei Laufwerk C ändern (US1 = 1 und US0 = 0). Deswegen muß man das Statusregister 0 meist zweimal lesen. Die untersten drei Bits dieses Registers zeigen dabei immer an, auf welches Laufwerk sich die Informationen in den restlichen Bits beziehen.

FDC – PRAXIS

Wem die theoretischen Ausführungen bis hier etwas zu trocken waren, für den ist das folgende Programm gedacht. Es zeigt, daß man auch von BASIC aus schon sehr viele Funktionen des FDC ansprechen kann.

Wenn man die Versuche hier nicht gerade mit eingelegter Lieblingsdiskette macht, kann eigentlich nichts zerstört werden. Etwas problematisch ist es allerdings, ein SEEK-Kommando zu Spuren ab 43 aufwärts zu starten. Hier macht der Schrittmotor etwas Krach, weil er dann in eine mechanische Sperre läuft.

Das Programm arbeitet nur mit Drive A, kann aber leicht erweitert werden.

Die Menü-Option z (result phase) ist so organisiert, daß automatisch alle noch anstehenden Bytes einer Ergebnisphase eingelesen und angezeigt werden. In Zeile 1030 sieht man, daß dazu ständig das Haupt-Statusregister gelesen wird: Die Bits 0 bis 3 werden ausgeblendet (FNs AND &F0) und dann getestet, ob die restlichen Bits so gesetzt sind, daß der FDC ein Byte aus der Result-Phase abgeben will: &D0 = &X11010000.

Option x (throw away) macht den FDC wieder frei, wenn man sich einmal verfranzt hat: In Zeile 1040 wird einfach das Datenregister so lange gelesen, bis das Haupt-Statusregister anzeigt, daß der FDC geneigt ist, wieder einen Befehl entgegenzunehmen. Es ist dabei ungefährlich, auch dann ein Byte vom FDC zu lesen, wenn dieser selbst eins vom Prozessor erwartet.

Die vom FDC angezeigte Datenrichtung (Bit 6 im Haupt-Statusregister) wird in dieser Routine einfach ignoriert.

```
 1 ' ***** FDC - Praxis *****
 2 ' by G.Woigk  vs. 27.03.86
 3 '
10 MODE 2
20 DEF FNs=INP(&FB7E)      ' Haupt-Statusregister einlesen
30 DEF FNd=INP(&FB7F)      ' Datenregister einlesen
35 DEF FNb$(z)=BIN$(z,8)   ' Ausdruck eines Bytes als 8 Bits
40 d=&FB7F                 ' Adresse Datenregister
50 m=&FA7F                 ' Adresse Motor
54 '
55 ' *** MENUE ***
56 '
60 PRINT "r - recalibrate
70 PRINT "t - track
80 PRINT "s - steuerregister
90 PRINT "1 - motor on
100 PRINT "0 - motor off
110 PRINT "i - statusregister 0
120 PRINT "z - result phase
130 PRINT "x - throw away
140 PRINT "d - statusregister 3
150 PRINT "? - ID lesen
490 '
491 ' *** Menueprogramm ***
492 '
500 WINDOW 41,80,1,25
510   i$=INKEY$:IF i$="" THEN 510
520   bef=INSTR("rts10izxd?",LOWER$(i$))
530   ON bef GOSUB 1070,1080,1020,1010,1000,1050,1030,1040,1060,1090
540 GOTO 510
590 '
591 ' *** Aufrufbare Unterprogramme ***
592 '
1000 OUT m,0:RETURN                ' 0 - Motor aus
1010 OUT m,1:RETURN                ' 1 - Motor ein
1020 PRINT "#";FNb$(FNs):RETURN    ' s - Steuerregister
1025 '
1030 IF (FNs AND &F0)=&D0 THEN PRINT">";FNb$(FNd):GOTO 1030 ' z - result phase
1040 WHILE (FNs AND &F0)<>&80:i=FNd:WEND:RETURN              ' x - trow away
1050 GOSUB 1040:PRINT"status 0":OUT d,8:GOTO 1030           ' i - Statusreg. 0
1060 GOSUB 1040:PRINT"status 3":OUT d,4:OUT d,0:GOTO 1030   ' d - Statusreg. 3
1065 '
1070 GOSUB 1040:OUT d,7:OUT d,0:RETURN                                ' Spur 0
1080 GOSUB 1040:INPUT "track ",t:OUT d,15:OUT d,0:OUT d,t:RETURN      ' Spur suchen
1090 GOSUB 1040:PRINT"ID":OUT d,74:OUT d,0:RETURN                     ' ID lesen
```

Der Schreib-/Lesespeicher – das RAM

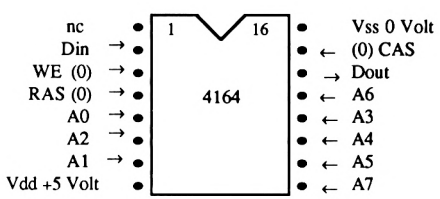
Das RAM des Schneider CPC 464 und 664 besteht aus 8 Bausteinen vom Typ 4164. Der Schneider CPC 6128 enthält entsprechend seiner doppelten Speicherkapazität 16 Stück davon.

Jedes IC hat genau $2^{16} = 65536$ verschiedene Speicherplätze, wovon jeder aus genau einem Bit besteht. Will die CPU also ein Byte lesen oder schreiben, muß an acht ICs dieselbe Adresse angelegt werden. Diese liefern dann zusammen die acht Bits, aus denen sich ein Byte zusammensetzt.

Alle Pins der RAM-Bausteine sind deshalb völlig parallel angeschlossen, und die jeweils entsprechenden Pins aller ICs sind miteinander verbunden. Ausgenommen sind natürlich die Datenanschlüsse, die jeweils mit einer anderen Leitung des Datenbusses verbunden sind.

Beim CPC 6128 sind sogar alle 16 ICs auf diese Weise völlig identisch angeschlossen. Nur die CAS-Anschlüsse (Column Address Strobe) sind für die beiden Bänke verschieden.

DIE ANSCHLUSSBELEGUNG DER RAMs 4164



Erklärung zu den Anschlüssen:

Vdd und Vss

Über diese Pins werden die RAMs mit Strom versorgt. Dynamische RAM-Bausteine verbrauchen kaum Strom. Allerdings müssen die Stromspitzen, die beim massenweisen Umschalten der Gatter in den ICs auftreten können, außen unbedingt durch parallel geschaltete Kondensatoren geglättet werden, sonst bricht im entscheidenden Moment immer die Spannung zusammen.

A0 bis A7 – Adreßleitungen

Um eine Speicherzelle korrekt zu adressieren, benötigt jedes IC genau 16 Adreßbits. Diese werden in zwei Hälften eingelatcht, wozu die Steuereingänge RAS und CAS dienen.

RAS und CAS – Row/Column Address Strobe

Mit einer negativen Flanke an RAS wird die obere Adreßhälfte (A8 bis A15 der CPU) vom RAM-IC übernommen. Diese nennt man auch, von der inneren Organisation der ICs her, Zeilenadresse. Die Spaltenadresse (A0 bis A7 der CPU) wird mit CAS übernommen.

Din und Dout – Data in/out

Die RAMs vom Typ 4164 haben einen getrennten Datenein- und -ausgang. Das ist speziell für sogenannte Read-Modify-Write-Zyklen gedacht, bei denen eine Speicherzelle gelesen und gleich darauf neu beschrieben wird. Die CPU Z80 ist dazu aber nicht in der Lage. Trotzdem wird im Schneider CPC von der Trennung des Datenein- und -ausgangs Gebrauch gemacht. Dout ist mit dem Datenbus der ULA verbunden, Din direkt mit dem der CPU.

WE – Write Enable

CAS dient gleichzeitig als Strobe (Signalimpuls), um das an Din anliegende Bit in die adressierte Speicherzelle zu übernehmen oder an Dout ein Bit auszugeben. Welche der Funktionen das RAM nun ausführt, wird durch den Pegel an WE bestimmt: Liegt WE an +5 Volt, so wird das RAM gelesen, das Bit also an Dout ausgegeben. Wird WE auf 0 Volt gezogen, kann das RAM beschrieben werden.

nc – not connected

Dieser Pin ist im RAM nicht angeschlossen. Die Nachfolge-ICs vom Typ 41256 mit vierfacher Speicherkapazität haben hier noch einen zusätzlichen Adreßanschluß.

REFRESH DER DYNAMISCHEN RAMS

Die Ansteuerung der dynamischen RAMs im Schneider CPC ist im Kapitel über die ULA bereits ausführlich behandelt worden. Dafür folgen hier ein

paar technische Details. Sie sollen erklären, wie das schier unersättliche Informationsbedürfnis von CPU und Video-Ausgabe (bis zu 3 Bytes pro Mikrosekunde) überhaupt realisiert werden konnte.

Zunächst wollen wir aber den Begriff "Refresh" erklären: Damit bezeichnet man das Wiederauffrischen der gespeicherten Informationen in dynamischen Speicherzellen. Diese speichern ihre Bits nämlich nicht statisch, also fest in Flip-Flops, wo eine Information bis zum Ausschalten des Computers unverändert erhalten bleibt.

Ihr Speicherprinzip ist "dynamisch": Um eine Eins zu speichern, wird in der entsprechenden Zelle ein Kondensator aufgeladen, für eine Null eben nicht. Aber der Kondensator verliert sehr schnell an Ladung, weil seine Kapazität nur einen Bruchteil Picofarads beträgt. Die gespeicherte Information ist auf dem besten Wege, verlorenzugehen.

Deshalb müssen alle Speicherzellen regelmäßig ausgelesen werden. Danach sind die Kondensatoren zwar völlig entladen. Schreibt man aber die gerade gelesene Information wieder in die Speicherzellen zurück, so ist der Inhalt wieder aufgefrischt.

Um die Informationen in einem dynamischen RAM zu erhalten, muß also ein erheblicher Aufwand betrieben werden. Der (entscheidende) Vorteil ist aber, daß pro gespeichertem Bit nur eine Gatterfunktion und ein Kondensator benötigt werden. Bei den statischen RAMs sind es 5 oder 6 Gatter! Dadurch kann dieselbe Information auf einem viel kleineren Chip gespeichert werden, was natürlich viel preiswerter ist.

Nun ist es aber ein sehr mühsames Geschäft, Zelle für Zelle aufzufrischen. Außerdem wäre man zu langsam, wollte man die einzelnen Speicherzellen nacheinander bearbeiten.

Man nutzt deshalb die Tatsache aus, daß die Speicherzellen auf dem Chip bereits in einer rechteckigen Fläche organisiert sind. Man liest deshalb immer eine ganze Zeile aus und schreibt deren Inhalt zurück.

Diesen Refresh führen alle Speicher-ICs automatisch durch, wenn man ihnen eine Adreßhälfte mit RAS (Row – Zeile) übergibt.

Das CAS-Signal selektiert dann nur noch eine spezielle Zelle innerhalb dieser Zeile, die dann ausgegeben (WE=1) oder neu beschrieben wird (WE=0).

Um also den Refresh aller Speicherzellen zu garantieren, müssen in jeder Sekunde mehrmals (ca. 50mal) alle Zeilenadressen abgearbeitet werden. Dabei

kommt man dem Z80 sogar so weit entgegen, daß für den Refresh nur sieben Bits der RAS-Adresse signifikant sind (das R-Register des Z80 umfaßt nur 7 Bit).

Trotzdem wird im Schneider CPC der Refresh von der ULA besorgt: Diese liest ja 50mal pro Sekunde den gesamten Bildschirmspeicher aus, der ein komplettes RAM-Viertel ausmacht. Die Adreßbits A14 und A15 haben einen konstanten Wert (und bestimmen damit, in welchem Adreßviertel der CPU der Bildwiederholspeicher liegt), A0 bis A13 werden aber komplett und immer wieder durchgegangen.

Daraus ergibt sich ein Problem: In der 8 Bit breiten RAS-Adreßhälfte bleiben die oberen beiden Bits konstant. Nur die unteren 6 Bits werden verändert. Somit würde nur die Hälfte des gesamten RAMs durch das Auslesen des Video-RAMs aufgefrischt.

Hier half man sich bei AMSTRAD mit einem Trick, der so simpel wie auch genial ist: Die Adreßleitungen der CPU werden einfach nicht an den entsprechenden Anschlüssen der dynamischen RAMs angeschlossen, sondern recht wild vertauscht.

Das ist ohne weiteres möglich, denn jeder Adreßanschluß der RAM-ICs ist eigentlich gleichwertig und kann willkürlich numeriert werden. Diese Numerierung kann, muß aber nicht die interne Struktur der ICs widerspiegeln.

Es ergeben sich nur zwei Einschränkungen. Die RAS- und CAS-Adreßhälften sind nicht beliebig untereinander austauschbar:

1. Die Adreßbits A14 und A15 der CPU sollten nicht in der RAS-Adreßhälfte angelegt werden, da das ja gerade das Problem mit dem Refresh verursacht.

2. A0 muß in der CAS-Hälfte liegen, da das Gate Array jeweils zwei Bytes im sogenannten Page Mode aus dem Video-RAM ausliest. Hierbei wird keine neue RAS-Hälfte adressiert, sondern für das zweite Byte nur noch CAS verändert.

Von den vielen möglichen Zuordnungen CPU-Adresse ⇔ RAM-Adresse hat man im Schneider CPC 464 folgende verwirklicht:

	RAS							CAS								
RAM	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
CPU	A2	A1	A4	A3	A6	A5	A8	A7	A9	A0	A11	A10	A13	A12	A15	A14

ADRESSIERUNGSARTEN DER DYNAMISCHEN RAM-ICS IM SCHNEIDER CPC

Der Zugriff auf ein spezielles Bit im Speicherfeld ist für die dynamischen RAMs ein äußerst komplizierter Vorgang.

Zunächst wird mit der fallenden Flanke an RAS (1-0-Übergang) die Zeilenadresse eingelesen. Sieben Bits davon werden wirklich benutzt, um eine Zeile im Speicherfeld auszuwählen. Das für den Refresh nicht signifikante Bit wird für die Spaltenadresse aufbewahrt.

Die so adressierte Speicherzeile, die aus 512 Bits besteht, wird in eine Hilfszeile statischen RAMs kopiert, worauf die Kondensatoren der dynamischen Speicherzeile leer sind!

Nun sollte möglichst bald der CAS-Impuls kommen, mit dem die Spaltenadresse eingelesen wird. Deren acht Bits und das von der Zeilenadresse übriggebliebene Bit wählen nun genau eine Zelle aus den 512 möglichen der Hilfszeile aus. Das dauert ein Weilchen; knapp 100 Nanosekunden nach der 1-0-Flanke an CAS ist das Bit aber angewählt.

Je nachdem, ob an WE ein logisches Null- oder Eins-Signal anliegt, wird es an Dout verstärkt ausgegeben oder entsprechend Din neu gesetzt.

Danach muß der Zwischenspeicher wieder in die Speicherzeile zurückkopiert werden, was mit der steigenden Flanke an RAS geschieht. Das Bit ist neu gesetzt (oder ausgelesen) und die Zeile wieder aufgefrischt.

Das Bit wird dabei an Dout so lange zwischengespeichert und ausgegeben, wie CAS auf Null-Potential liegt.

Das ist der einfachste Fall, ein ganz normaler Schreib- oder Lesezugriff, wie er im Schneider CPC immer für die CPU ausgelöst wird. Man erkennt aber schon, daß bestimmte Wartezeiten eingehalten werden müssen, bis die Information jeweils bereitsteht. Und vor allem: Die "Zugriffszeit" auf ein Datenelement ist nicht die gleiche wie die "Zykluszeit": Nach dem "Zugriff" auf das Datenelement muß der "Zyklus" noch beendet werden, indem die Hilfszeile in den Speicher zurückkopiert wird.

Die im Schneider CPC verwendeten RAMs haben eine Zugriffszeit von 150 Nanosekunden. Ihre minimale Zykluszeit beträgt 260 Nanosekunden. Damit wären theoretisch 3 Speicherzugriffe pro Mikrosekunde möglich: 3 * 260 = 780. Das setzt allerdings eine optimale Abfolge der einzelnen Signale voraus, die sich in der Praxis nur selten erreichen läßt.

Im Schneider CPC wird deshalb auch vom sogenannten Page Mode Gebrauch gemacht: Die ULA liest pro Mikrosekunde zwei Bytes aus dem Video-RAM, die sich nur im untersten Adreßbit (A0) unterscheiden. Dafür muß keine neue Zeile in den RAM-ICs angewählt werden. Ein neuer CAS-Impuls mit einer neuen Spaltenadresse genügt. Ein Page-Mode-Lesezyklus ist dabei nur 125 Nanosekunden lang. Die für die drei Speicherzugriffe benötigte Zeit reduziert sich also auf 260 + 260 + 125 = 625 Nanosekunden. Das ist natürlich wieder nur der Idealwert. Die einzelnen Signale, die die ULA aus dem 16-MHz-Eingangstakt erzeugen muß, lassen sich jetzt aber schon recht großzügig bis zur nächsten Flanke des Taktes ausweiten.

Die Festwertspeicher – die ROMs

Alle Programme, die der Schneider CPC beim Einschalten schon bereitstellt, sind in ROMs gespeichert. Hierbei handelt es sich um Speicher-ICs, deren Inhalt beim Ausschalten nicht verlorengeht, dafür aber bei eingeschalteter Versorgungsspannung auch nicht mehr verändert werden kann. Daher die Bezeichnung ROM (Read Only Memory – Nur-Lese-Speicher oder Festwertspeicher).

Der Schneider CPC 464 hat bereits 32 kByte Festwertspeicher; die CPCs 664 und 6128 48 kByte. Diese Kapazität erreicht der CPC 464 ebenfalls, wenn man einen Diskettenkontroller hinten an den Systembus ansteckt.

In den ROMs ist das Betriebssystem fest programmiert. Dieser Teil (16 kByte) wird bei Bedarf mittels Bankswitching immer im untersten Adreßviertel der CPU eingeblendet.

Es enthält die Reset-Routine, die unter anderem beim Einschalten des Computers abgearbeitet wird, Routinen für den Ticker-Interrupt, Bankswitching und, nach Abteilungen geordnet, alle programmtechnischen Schnittstellen zur Hardware des Computers: Zeichen-, Grafik-, Tonausgabe, Tastaturverwaltung und ähnliches mehr.

Alle weiteren ROM-Bänke werden im obersten Adreßviertel eingeblendet. Hierbei handelt es sich in erster Linie um den BASIC-Interpreter, der die eingegebenen BASIC-Programme abarbeitet und um das AMSDOS-ROM.

Dort sind nicht nur die Treiberroutinen für die Diskettenlaufwerke untergebracht, sondern noch einiges mehr: Programme für eine SIO (serielle Schnittstelle), die aber wahrscheinlich nie benutzt werden (die von Schneider nun angebotene RS-232-Schnittstelle läßt sich damit nicht betreiben); die Boot-

Routine für CP/M und 8 kByte von DR LOGO. Logo wird zwar größtenteils auch von Diskette geladen. Hätte man aber nicht einen Teil in das AMSDOS-ROM verbannt, bliebe im RAM ziemlich genau kein einziges Byte mehr für LOGO-Prozeduren übrig.

Die ROMs werden wie die ULA speziell für AMSTRAD hergestellt. Sie haben aber dennoch einen Hauch von Serienmäßigkeit: Das elektronische Layout ist Sache der Herstellerfirma. Nur der Inhalt der Speicherzellen wird von AMSTRAD bestimmt. Dafür wird für jede ROM-Serie eine sogenannte "Maske" angefertigt, die dann bereits bei der Produktion über gesetzte und nicht gesetzte Bits entscheidet.

Betriebssystem und BASIC-Interpreter sind zusammen in einem 32-kByte-ROM untergebracht. Da in jedem neuen CPC auch ein neues Betriebssystem und bei den 6er-Typen auch ein erweiterter BASIC-Interpreter enthalten ist, haben alle ein anderes ROM. Die Unterschiede liegen aber, wie gesagt, nur in der Maske und äußern sich in unterschiedlichen 40-Tausender-Nummern, die auf die ICs aufgedruckt sind.

Das 32-k-ROM ist vom Grundtyp 23256 und soweit Pin-kompatibel wie überhaupt möglich mit den Eproms der Serie 27256. Das AMSDOS-ROM mit nur 16 kByte Speicherkapazität hat die Typenbezeichnung 23128 und ist Pin-kompatibel mit den 27128-Eproms. Von der Beschaltung her ist das AMSDOS-ROM im Ansteck-Controller für den Schneider CPC 464 sofort gegen ein Eprom austauschbar. Hier ist der Pin 1, Vpp (Programmierspannung), an die Versorgungsspannung Vcc von +5 Volt angeschlossen, wie das für den normalen Lesebetrieb bei Eproms vorgeschrieben ist.

Alle anderen ROMs können durch entsprechende Eproms ersetzt werden, wenn man einen kleinen Eingriff an der Platine vornimmt: Hier ist Pin 1 nicht beschaltet und muß erst mit einer kleinen Drahtbrücke an +5 Volt angeschlossen werden.

Freundlicherweise wurde auch im CPC 664 und 6128 am AMSDOS-ROM der Pin 27 für den Programmierimpuls "lesefertig" an +5 Volt angeschlossen, obwohl das für das ROM sicher auch nicht notwendig war.

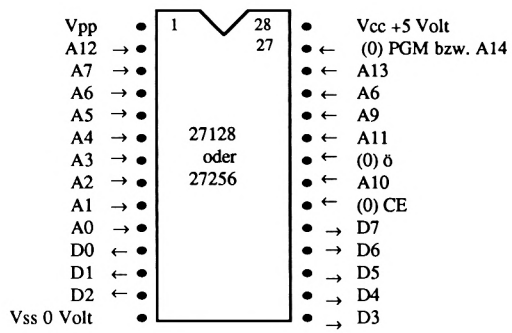
Man kann die Eproms aber auch außen auf einer Erweiterungsplatine anstecken. Es ist nämlich möglich, alle ROMs im Schneider CPC mit der Signalleitung ROMDIS von außen her abzustellen. Das hat den Vorteil, daß man bei Bedarf auch weiterhin die Original-Firmware zur Verfügung hat und eventuelle Garantie-Ansprüche nicht erlöschen. Nachteil ist aber der erheblich höhere Aufwand.

Ein Umstricken der Firmware gestaltet sich dabei gar nicht so teuer, wie man annehmen mag: Ein 16-kByte-Eprom ist bereits für unter 10 DM zu haben, eins mit 32 kByte für deutlich unter 20 DM. Teurer ist dagegen ein Programmiergerät.

Da es für den Anwender sicher viel interessanter ist, Anschlußbelegung und Funktionsweise der Eproms zu erfahren, sind diese in den folgenden Schaubildern dargestellt. Die ROMs sind vollkommen identisch, nur fallen hier die Anschlüsse für die Programmierung weg (Pin 1 und beim 23128 auch Pin 27).

Da sich die Anschlußbelegung weiterhin zwischen 16- und 32-kByte-(EP)ROM nur um Pin 27 unterscheidet, werden diese beiden Typen zunächst zusammen behandelt, und erst zum Schluß wird auf die Unterschiede (vor allem bei der Programmierung) eingegangen.

ANSCHLUSSBELEGUNG DER EPROMS 27128 UND 27256



Erklärung zu den Anschlüssen:

Vcc, Vss und Vpp

Über Vcc und Vss erfolgt die Stromversorgung der ROMs. Die Eproms müssen im normalen Lesebetrieb auch an Pin 1, Vpp an +5 Volt angeschlossen werden. Beim Programmieren wird an Vpp die wesentlich höhere Programmierspannung angelegt. Pin 1 muß bei den ROMs nicht beschaltet sein. Er ist hier intern nicht angeschlossen.

PGM – Program

Pin 27 dient beim 16-kByte-Eprom 27128 als Steuereingang für die Programmierspannung. Beim 16-kByte-ROM (AMSDOS) müßte dieser Eingang eigentlich nicht beschaltet sein. Er ist aber Eprom-kompatibel an Vcc angeschlossen.

OE und CE – Output Enable und Chip Enable

Nur wenn beide Eingänge auf Null-Pegel liegen, geben die Eproms auf ihren Datenleitungen ein Byte aus. OE sollte dabei normalerweise direkt mit RD des Prozessors verbunden sein und CE mit einer Adreßdecodierung für das Eprom. Liegt an CE eine positive Spannung an, befindet sich das Eprom im sogenannten Stand-by-Modus, wo der Stromverbrauch auf weniger als die Hälfte gedrosselt wird. Im Schneider CPC ist CE an ROMEN vom Gate Array angeschlossen und OE an den Anschluß ROMDIS des Expansion Ports.

A0 bis A14 – Adreßleitungen

Diese Leitungen sind direkt mit dem Adreßbus der CPU verbunden und wählen bei einem Lesezugriff auf das ROM die Speicherzelle aus.

D0 bis D7 – Datenleitungen

Die Datenleitungen sind ebenfalls direkt mit dem Datenbus der CPU verbunden.

PROGRAMMIERUNG DER EPROMS

Die in einem Eprom gespeicherten Daten können, im Gegensatz zum ROM, sowohl programmiert als auch gelöscht werden. Das Programmieren geschieht elektrisch; gelöscht werden Eproms mit ultraviolettem Licht. Wer die Anschaffung eines Eprom-Löschgerätes scheut, kann sie einfach an die Sonne legen. Je nach Bestrahlungsintensität dauert das aber ein paar Wochen. Auch eine Höhensonne ist mitunter geeignet.

Zum Programmieren wird eine "Programmierspannung" benötigt, die weit über der normalen Versorgungsspannung liegt. Beim 27128 beträgt sie 21 Volt, beim 27256 nur 12.5 Volt. Ein komplett gelöschtes Eprom enthält in allen Speicherzellen ein gesetztes Bit (1). Wenn man ein Eprom ausliest, muß

man also überall den Wert 255 erhalten. Programmiert wird, indem einzelne Bits auf 0 gesetzt werden.

Dazu legt man an die Adreßleitungen die Adresse einer Speicherzelle an und an die Datenleitungen, die nun als Eingang fungieren, das gewünschte Datenbyte. Dann werden die im Byte enthaltenen Nullen hineingepulst.

Man kann hier immer mit der maximal zulässigen Dauer arbeiten (wodurch das Programmieren eines Eproms eine recht langwierige Angelegenheit wird) oder intelligenter vorgehen: Man erzeugt so lange kurze Programmierimpulse, bis der Wert, den man aus dieser Speicherzelle liest, dem gewünschten entspricht (d.h. bis alle Nullen programmiert sind). Dann schickt man noch einen längeren Impuls hinterher, damit das Datenelement auch "wirklich sicher sitzt".

Die Programmierspannung Vpp darf dabei nur angelegt werden, wenn das Eprom auch über Vcc mit Strom versorgt wird. Also: Vpp erst nach Vcc anlegen, und Vpp nachher wieder vor Vcc abklemmen.

Das Brennen eines 16-kByte-Eproms unterscheidet sich verhältnismäßig stark vom Brennen der 32-kByte-Typen. Zum einen muß man auf die unterschiedliche Programmierspannung achten, zum anderen wird der Programmierimpuls beim 27128 über den Anschluß PGM gesteuert, der beim 27256 aber für A14 herhalten muß. Hier wird der Impuls deshalb mit CE erzeugt.

Man muß deshalb der Treibersoftware mitteilen, was für einen Typ man zu brennen gedenkt (wenn man sich nicht ausschließlich auf 16-kByte-Eproms spezialisiert). Das Programm muß dann die einzelnen Anschlüsse des Eproms unterschiedlich ansteuern.

Es besteht aber auch die Möglichkeit, diese Information aus dem Eprom selbst zu erfahren: Wird A9 auf +12 Volt gelegt (bei beiden Typen!), so geben diese auf dem Datenbus einen Code für ihren Hersteller heraus (wenn A0 auf Null liegt) und ihre Größe (bei A0 = 1).

Alle anderen Adreßbits müssen bei der Aktion auf logisch Null gehalten werden.

Identifikationscode:

27128 - &83
27256 - &04

27256 – ARBEITSMODI

Die folgenden Tabellen geben einen Überblick über die Zustände an den Steuereingängen, die bei den Eproms 27128 und 27256 erlaubt sind:

CE	OE	A9	Vpp	D0-D7 / Modus
0	0	x	Vcc	Data out, normaler Lesezugriff
0	1	x	Vcc	hohe Impedanz, Output Disable
1	x	x	Vcc	hohe Impedanz, Stand By
0	0	12V	Vcc	Identifikationscode &04 (wenn A0=1)
0	1	x	Vpp	Data in, Programmieren
1	1	x	Vpp	hohe Impedanz, neutraler Zustand beim Programmieren
x	0	x	Vpp	Data out, Verify (Achtung bei NEC-Typen: s.u.)

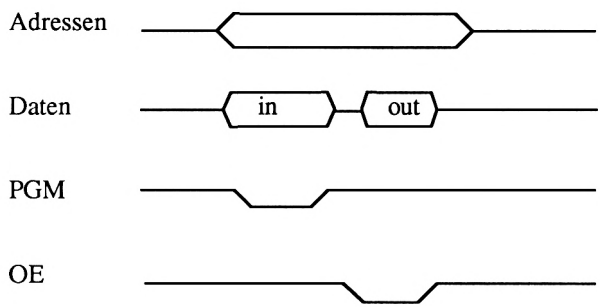
27128 – ARBEITSMODI

CE	OE	PGM	A9	Vpp	D0-D7 / Modus
0	0	1	x	Vcc	Data out, normaler Lesezugriff
0	1	1	x	Vcc	hohe Impedanz, Output Disable
1	x	x	x	Vcc	hohe Impedanz, Stand By
0	0	1	12V	Vcc	Identifikationscode &83 (wenn A0=1)
0	1	0	x	Vpp	Data in, Programmieren
0	0	1	x	Vpp	Data out, Verify
0	1	1	x	Vpp	hohe Impedanz, neutraler Zustand beim Programmieren
1	x	x	x	Vpp	hohe Impedanz, neutraler Zustand beim Programmieren

Die nun folgenden Tabellen zeigen in Zeitdiagrammen, wie die Programmierung eines 27128- oder 27256-Eproms abläuft. Vpp liegt dabei die ganze Zeit auf der jeweiligen Programmierspannung, und Vcc sollte nach Möglichkeit auf +6 Volt erhöht werden.

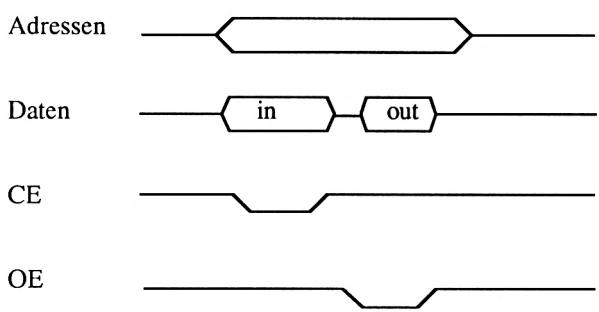
TIMING-DIAGRAMM "PROGRAMMING 27128"

Vpp liegt die ganze Zeit auf +21 Volt, und CE ist aktiv (0 Volt).



TIMING DIAGRAMM "PROGRAMMING 27256"

Vpp liegt die ganze Zeit auf +12.5 Volt.



Vorsicht ist allerdings bei 27256-Eproms von NEC geboten: Hier muß beim Verify nicht nur OE, sondern unbedingt auch CE auf 0 Volt gelegt werden. Diese Eproms reagieren (zumindest die aktuellen Typen) sonst mit Selbstzerstörung. Am besten ist es hier, statt einem Verify einen ganz normalen Lesezugriff zu starten (mit +5 Volt an Vpp).

Die Dauer eines Programmier-Impulses, die über PGM bzw. CE gesteuert wird, hängt vom gewählten Verfahren ab. Bei der langsameren Methode ist der Impuls immer 50 Millisekunden (ms) lang. Damit schafft man gerade 20 Bytes pro Sekunde. Für den 27128 braucht man also eine knappe Viertelstunde, für den 27256 sogar eine halbe Stunde!

Erheblich schneller geht es aber, wenn man jedes Byte nur so lange programmiert, bis es "sitzt". Die Firma Intel schlägt für ihre Eproms folgendes Verfahren vor:

27128: Bis zu 15 kurze Impulse von je 1 ms, bis das Datenelement sitzt; danach einen langen Impuls mit der Länge: 4 ms * Anzahl Impulse.

27256: Bis zu 25 kurze Impulse von je 1 ms, bis das Datenelement sitzt; danach einen langen Impuls mit der Länge: 3 ms * Anzahl Impulse.

Die Schnittstellen der Schneider CPC-Rechner

DIE STROMVERSORGUNG

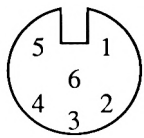
Die Anschlüsse für die Stromversorgung sind bei den drei CPC-Rechnern weitgehend gleich ausgefallen. Beim Schneider CPC 664 und 6128 kommt nur eine zusätzliche Verbindungsleitung mit "umgekehrter Logik" (Kabel am Computer) für die 12-Volt-Versorgung des Floppy-Laufwerks hinzu.

Identisch ist bei allen drei Typen die 5-Volt-Versorgung. Hier dürfte nur der CPC 464 erheblich weniger Strom ziehen, weil er keinen Disk-Controller und kein Laufwerk mitzuversorgen hat. Das ändert sich natürlich in dem Moment, wo man den Kassettenrecorder einschaltet. In beiden Fällen darf der 5-Volt-Ausgang des Monitors nicht mit mehr als 2 Ampère belastet werden.

DAS MONITORSIGNAL

An der Monitorbuchse werden alle Signale bereitgestellt, um entweder einen monochromen oder einen Farbmonitor zu betreiben. Sehr viele Standardmonitore können direkt angeschlossen werden, wenn man einmal von der Stromversorgung absieht, die dann getrennt besorgt werden muß. Wenn man sich von irgendwoher über einen Vorwiderstand auch noch +12 Volt holt (was beim CPC 664 und 6128 ja nicht besonders schwer ist), so kann man sogar einen Fernseher mit Scartbuchse anschließen. Die 12 Volt werden dabei ausschließlich gebraucht, um den Fernseher auf Monitorbetrieb umzuschalten.

Bei der Monitorbuchse am Schneider CPC handelt es sich um eine sechspolige DIN-Buchse. Die Pins sind dabei wie folgt durchnumeriert (Aufsicht von außen auf die Buchse):



Wer sich aus irgendeinem Grund ein Adapterkabel basteln will: Auf jedem Stecker sind die Nummern neben jedem Pin eingeprägt. Damit sollte kaum noch Verwirrung auftreten.

Die einzelnen Anschlußstifte haben folgende Funktion:

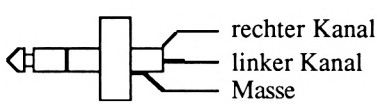
1 – R = Helligkeit des Rot-Anteils
2 – G = Helligkeit des Grün-Anteils
3 – B = Helligkeit des Blau-Anteils
4 – Synchronisation für den Strahlrücklauf (horizontal & vertikal)
5 – Referenz = Masse-Anschluß 0 Volt
6 – Mischsignal aus 1, 2, 3, und 4 = Luminanz (einige monochrome Monitore)

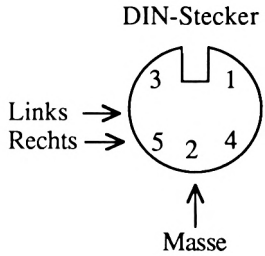
DER AUDIO-ANSCHLUSS

An diesem Anschluß werden die drei Kanäle des PSG (Sound-Chip) leicht vermischt ausgegeben. Der Anwender sollte dieses Angebot der AMSTRAD-Designer unbedingt annehmen und sich ein Verbindungskabel zu seiner Stereoanlage basteln. Dabei ist es uninteressant, ob der heimische Adapterstandard DIN oder CINCH ist, in beiden Fällen ist ein Anschluß möglich. Man sollte sich einen unbenutzten Eingang am Verstärker aussuchen, der für einen Tuner (Radio), Kassettenrecorder oder einen Plattenspieler mit Kristall-Tonabnehmer vorgesehen ist.

Der Stecker selbst ist ein 3,5mm-Klinkenstecker Stereo und muß wie folgt angeschlossen werden:

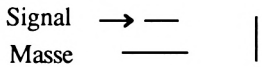
Klinkenstecker am Computer





Cinch-Stecker

(Es werden zwei Stecker für den linken und für den rechten Kanal benötigt)



Wer nur einen Monoverstärker zur Verfügung hat, kann auch gefahrlos beide Kanäle zusammenmischen, indem er die beiden Signalleitungen zusammenlötet. Beim DIN-Stecker muß dann Pin 3 als Mono-Eingang verwendet werden.

Die drei Kanäle des PSG werden dabei wie folgt verteilt:

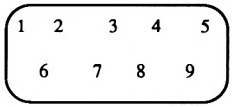
Rechts = A + B/2
Links = C + B/2

DER JOYSTICK-ANSCHLUSS

Bis auf eine kleine Abweichung entspricht der Joystick-Stecker völlig der Norm, so daß man jeden handelsüblichen Joystick sofort anschließen kann. Eine Maus kann nicht angeschlossen werden, weil hierfür sowohl eine echte Masse als auch eine Versorgungsspannung fehlt.

Die "kleine Abweichung" ist die Tatsache, daß an diese eine Buchse zwei Joysticks angeschlossen werden können. Es gibt in der Buchse zwei Nulleiter: einen für den ersten Joystick und einen für den zweiten. Wer eigene Bastelarbeiten scheut, aber dennoch zwei Joysticks braucht, muß als ersten Joystick das Original-Schneider-Exemplar erwerben. Dieser hat noch einen Ausgang, an dem jetzt der zweite Nulleiter an der Stelle des ersten herausgeführt ist.

Ansicht von hinten auf die Buchse:



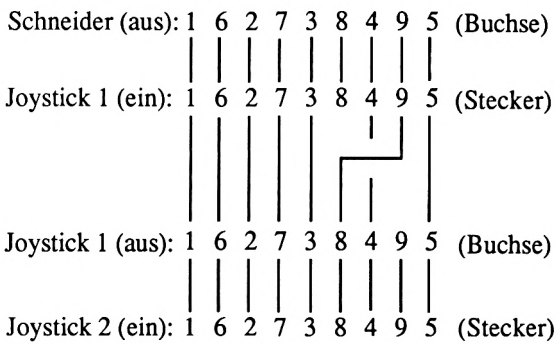
Belegung der einzelnen Stifte:

(Sichtweise der Richtungsangaben: von den Joysticks aus)

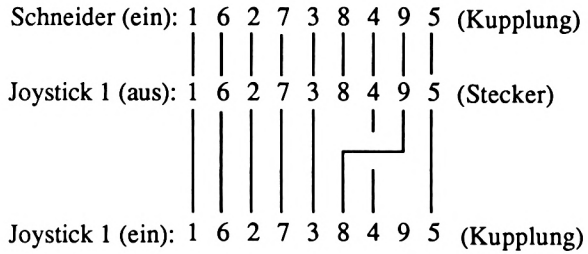
1 – Schalterausgang für UP (nach oben)
2 – Schalterausgang für DOWN (nach unten)
3 – Schalterausgang für LEFT (nach links)
4 – Schalterausgang für RIGHT (nach rechts)
5 – Schalterausgang (nicht spezifiziert)
6 – Schalterausgang für Feuer 2
7 – Schalterausgang für Feuer 1
8 – Gemeinsamer Eingang für alle Schalter von Joystick 1 (COMMON 1)
9 – Gemeinsamer Eingang für alle Schalter von Joystick 2 (COMMON 2)

Besonders Pin 5 ist interessant, da er in der gesamten Schneider-Literatur immer nur mit "nicht belegt" angegeben ist. Würde dieser Eingang an Joystick 1 irgendwie betätigt (Feuer 3 o.ä.) so erhielte man die Taste mit der Nummer 78, die angeblich nirgends angeschlossen ist. Mit Joystick 2, der ja parallel zu Tasten der normalen Tastatur angeschlossen ist, ergäbe sich Taste &36 = B.

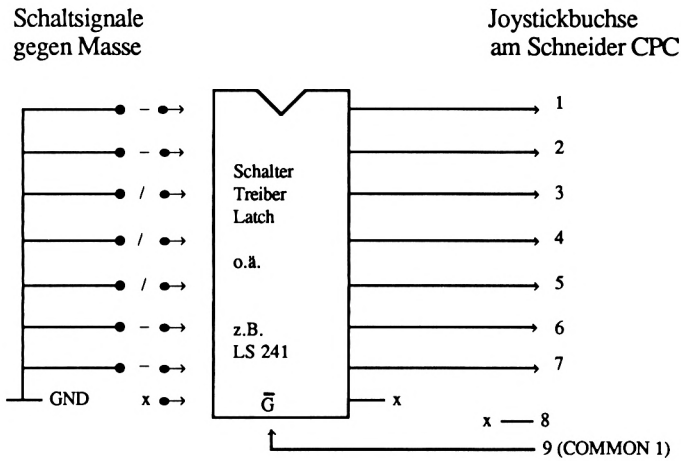
Anschluß von zwei Joysticks, Schneider-Methode:



Selbstlöt-Methode (Schaltung in einem Y-Adapter):



Die Joysticks sind beim Schneider CPC an die Tastaturmatrix angeschlossen. Joystick 1 auf eigenen Positionen, Joystick 2 parallel zu Tasten des Haupttastenfeldes. Die gemeinsamen Schaltereingänge sind identisch mit einem Zeilendraht der Matrix. Die Schalterausgänge sind mit sieben verschiedenen Spaltendrähten verbunden.



Aus dieser Beschaltung ergibt sich, daß es nicht zulässig ist, die Eingänge der Joystick-Buchse gegen Masse zu schalten, wenn man beispielsweise eine Zusatzschaltung anschließt, um Daten o.ä. zu übertragen. Dann würde sofort die Tastaturabfrage nicht mehr funktionieren. In diesem Fall müßte man praktisch einen Bustreiber zwischenschalten und Leitung 8 (COMMON 1) mit dessen Chip-Enable-Eingang verbinden. An diesen Eingängen können dann die logischen 0- oder 1-Pegel statisch anliegen. Auf die Spaltenleitungen (Pins 1 bis 7

der Buchse) werden sie aber nur durchgeschaltet, wenn die Tastatur auf der richtigen Zeile (0-Signal an COMMON 1 = Pin 8) abgefragt wird. Durch Einsatz von zwei ICs, von denen das eine von COMMON 1 und das andere von COMMON 2 durchgeschaltet wird, können sogar 14 Eingangssignale verarbeitet werden.

COMMON 1 und 2 entsprechen den Zeilendrähten 9 und 6 der Tastaturmatrix. Hoch, Runter, Links, Rechts, Feuer 2, Feuer 1 und Pin 5 (nicht spezifiziert) entsprechen den Spaltendrähten (bzw. Bits in den eingelesenen Bytes) 0 bis 6 in dieser Reihenfolge. Die Tastennummern berechnen sich jeweils als 8 * Zeile + Spalte und ergeben so die im Anhang angegebenen Werte für die Joystick-Tasten.

COMMON 1 – Zeile 9 → 8 * 9 = 72
COMMON 2 – Zeile 6 → 8 * 6 = 48

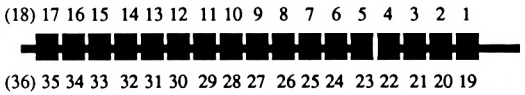
		Joystick 1	Joystick 2
HOCH	– Spaltenbit 0	0+72 = 72	0+48 = 48 (Taste 6)
RUNTER	– Spaltenbit 1	1+72 = 73	1+48 = 49 (Taste 5)
LINKS	– Spaltenbit 2	2+72 = 74	2+48 = 50 (Taste R)
RECHTS	– Spaltenbit 3	3+72 = 75	3+48 = 51 (Taste T)
FEUER 2	– Spaltenbit 4	4+72 = 76	4+48 = 52 (Taste G)
FEUER 1	– Spaltenbit 5	5+72 = 77	5+48 = 53 (Taste F)
PIN 5	– Spaltenbit 6	6+72 = 78	6+48 = 54 (Taste B)

DER DRUCKER-PORT

Für den Anschluß eines Druckers hat man sich beim Schneider CPC für den Industriestandard entschieden und eine Parallelschnittstelle nach Centronics-Norm implementiert. Beim CPC 464 und 664 ist der Anschluß einfach ein Platinenstecker. Das heißt, die Rechnerplatine ist an dieser Stelle einfach etwas verlängert und hat unten und oben Leiterbahnen aufgeätzt, die zum Schutz vor Korrosion verzinnt sind. Beim CPC 6128 hat man sich für die teurere Lösung entschieden und auch die mechanische Verbindung der Centronics-Norm angepaßt. Daß hier überhaupt statt der "gewohnten" Platinenstecker plötzlich drei aufwendige Buchsen die Rückseite der Rechnerkonsole zieren, daran ist die Post schuld, die jetzt auch die Computer als Verursacher störender Hochfrequenzstrahlung entdeckt hat und entsprechend auf allumfassende Weißblechabschirmung pocht.

Der Anschluß eines Druckers an den CPC 6128 ist entsprechend einfach: Der Drucker wird mit dem Verbindungskabel einfach angeschlossen. Wer aber

einen CPC 464 oder 664 besitzt, muß sich vorher mit der Anschlußbelegung auseinandersetzen. Die hat vor allem bei den "Platinenstecker-Typen" einen kleinen Haken: Um standardisierte Steckverbinder benutzen zu können, hat man hier ein Anschlußpaar am Computer wegrationalisiert. Ein Centronics-Stecker hat 36 Anschlüsse, die Platinenstecker aber nur 34. Dadurch gehen aber keine Funktionen verloren, denn schon eine Verbindung mit nur 22 Leitungen umfaßt alle verwendeten Signale. Wer also am Kabel sparen will, kann das gefahrlos tun.



Diese schematische Darstellung zeigt den Platinenstecker des CPC 464 oder 664 in der Aufsicht von hinten. Das Anschlußpärchen 18/36 existiert nicht, und zwischen Pin 4 und 5 (bzw. 22 und 23) ist die Platine ausgefräst, um einen Verpolungsschutz realisieren zu können. Die Centronics-Buchse des CPC 6128 unterscheidet sich nur in soweit, als hier die Anschlüsse 18 und 36 existieren.

Diese Anschlüsse sind wie folgt belegt:

1 ———	Strobe (0)	GND – Ground – Masse
19 –	GND	n.c. – not connected – nicht angeschlossen
2 ———	Data Bit 0	
20 –	GND	
3 ———	Data Bit 1	
21 –	GND	
4 ———	Data Bit 2	
22 –	GND	
5 ———	Data Bit 3	
23 –	GND	
6 ———	Data Bit 4	
24 –	GND	
7 ———	Data Bit 5	
25 –	GND	
8 ———	Data Bit 6	
26 –	GND	
9 ———	Data Bit 7 (GND)	
27 –	n.c.	
10 ———	n.c. (Acknowledge)	
28 –	GND	

11	<——	BUSY (1)	
	29 –	n.c.	
12	———	n.c.	
	30 –	n.c.	
13	———	n.c.	
	31 –	n.c.	
14	———	GND	
	32 –	n.c.	
15	———	n.c.	
	33 –	GND	
16	———	GND	
	34 –	n.c.	
17	———	n.c.	
	35 –	n.c.	
18	———	n.c.	(existiert beim Platinenstecker nicht)
	36 –	n.c.	(existiert beim Platinenstecker nicht)

Wer, was zu empfehlen ist, für die Verbindung Flachbandkabel und -stecker benutzt, hat im Kabel die einzelnen Signale in der Reihenfolge vorliegen, wie sie in der vorhergehenden Tabelle aufgelistet sind. Die einzelnen Leitungen bilden dabei immer Pärchen, wobei der Anschluß auf der Unterseite (n+18) als Masseleitung und Abschirmung für den entsprechenden Anschluß auf der Oberseite (n) dienen soll. Die Numerierung der Unterseite von 19 bis 36 bezieht sich auf den Centronics-Stecker. Der Platinenstecker, der beim CPC 464 bzw. 664 benutzt werden muß, hat, da er ja um ein Signalpaar kürzer ist, auf der Unterseite die Numerierung von 18 bis 34, also immer um 1 kleinere Angaben.

Wer nur ein 24-adriges Kabel verwendet, hat davon eventuell sogar einen Vorteil: Da die Leitungen 14 und 16 auf Masse gelegt sind, erzeugen viele Drucker nach dem Wagenrücklaufzeichen CHR$(13) automatisch einen Zeilenvorschub. Der Schneider CPC sendet aber nach jeder Zeile automatisch zwei Steuerzeichen: CHR$(13) und CHR$(10). Dadurch gehen einige Drucker recht verschwenderisch mit dem Papier um: Pro Zeile ein doppelter Vorschub ergibt nur noch ca. 30 statt 60 Textzeilen auf jedem Blatt.

Wenn dieses Problem bei Ihnen auftritt, haben sie viele Möglichkeiten, es abzustellen:

1. Verwenden sie nur ein 24-adriges Kabel.
2. Unterbrechen Sie die entsprechenden Leitungen im Kabel oder auf der Platine des Rechners.
3. Schauen Sie im Handbuch des Druckers nach. Die Drucker besitzen alle 8 bis 16 DIP-Schalter. Irgendeiner davon ist für den zusätzlichen Zeilenvorschub verantwortlich und muß umgeschaltet werden.

4. Bei CPC 664 und 6128 können Sie das Steuerzeichen 10 (LF) in die Printer-Translation-Table eintragen, um es fürderhin ignorieren zu lassen. Dafür müssen Sie sich aber der Maschinensprache bedienen. Diese Methode hat allerdings, wie ein Patchen von IND MC WAIT PRINTER, den Nachteil, daß davon auch die Grafikausgabe betroffen ist, was beim Ausdruck eines Bildschirmabzugs unangenehme Folgen haben kann.

Noch ein letztes Problem zeichnet diese Druckerschnittstelle aus: Wie in der vorhergehenden Tabelle bereits angedeutet, ist das höchstwertige Datenbit (D7) einfach auf Masse gelegt. Das hat zur Folge, daß die Grafikausgabe bei den meisten Druckern unnötig erschwert wird und teilweise überhaupt nicht möglich ist. Auch die Ausgabe von Sonderzeichen wird so erschwert oder ganz unmöglich gemacht. Dadurch, daß das höchstwertige Bit auf 0 gelegt ist, lassen sich nur noch die Zeichen mit den Codes 0 bis 127 ausdrucken. Von größeren Codes (128 bis 255) wird immer 128 abgezogen (&X10000000 = 128). Da der ASCII-Zeichensatz nur als 7-Bit-Code genormt ist, ist eine solche Verstümmelung dieser Schnittstelle noch normgerecht. Standard ist aber ein Centronics-Port mit 8 Datenleitungen.

Die Ursache für das fehlende achte Datenbit liegt an der Sparsamkeit der Hardware-Designer bei AMSTRAD. Der Datenbus der CPU umfaßt 8 Bits. Wieso wurde D7 (das höchstwertige Datenbit) nicht, wie alle anderen, zur entsprechenden Leitung des Drucker-Anschlußes durchgeführt? Die Antwort bringt ein Blick auf den Schaltplan:

Mit Bit 7 wird das Strobe-Signal erzeugt! Der Datenbus der CPU ist mit einem Achtfach-Daten-Latch (Speicher/Treiber), ein 74LS273, von den Datenleitungen zum Drucker abgekoppelt. Die Werte vom Datenbus werden übernommen, wenn A12 und IOWR aktiv werden. Letzteres ist ein Signal, das bereits an anderer Stelle aus IORQ und WR erzeugt wurde. Der Befehl OUT & EFFF,zeichen schreibt das Byte "zeichen" in das Daten-Latch ein.

Nur die Bits D0 bis D6 werden an den Drucker weitergeleitet. D7 ist über einen zusätzlichen Inverter mit der Strobe-Leitung verbunden.

Um ein Byte zum Drucker zu senden, muß man das Zeichen also insgesamt dreimal in das Daten-Latch schreiben: einmal mit zurückgesetztem Bit 7 (dadurch erscheint am Ausgang des Inverters ein logischer Eins-Pegel, also "kein Strobe"), dann mit gesetztem Bit 7 (Strobe) und dann wieder mit zurückgesetztem Bit 7.

Bei den "Hardware-Basteleien" findet sich aber ein Bastelvorschlag, wie Sie Ihrem CPC mit einem einfachen Stück Draht doch noch zu einem achten Bit verhelfen können.

DER EXPANSION-PORT (SYSTEMBUS)

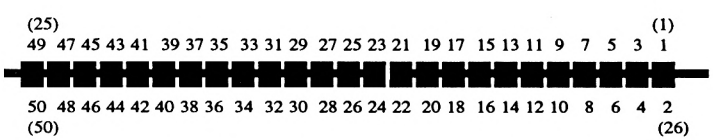
Der größte Anschluß an der Rückseite des Computers ist mit 50 Kontakten die als Expansion-Port bezeichnete Verbindung zum Rechner-internen Systembus. Hier angesteckte Erweiterungen werden logisch zu einem integralen Bestandteil des Computers. Diese Hardware könnte genausogut auf der Rechnerplatine selbst untergebracht sein. Das erkennt man beispielsweise am AMS-DOS-Diskettenkontroller, der beim Schneider CPC 464 hinten am Systembus angeschlossen wird und beim CPC 664 und 6128 bereits auf der Rechnerplatine integriert ist. Alle komplizierteren Erweiterungen werden am Expansion-Port angeschlossen: Modulbox, serielle Schnittstelle, Eprom-Brenner oder auch der Diskettenkontroller.

Beim CPC 464 und 664 ist diese Verbindung als Platinenstecker ausgeführt, beim Schneider CPC 6128 aus den weiter oben erwähnten Gründen als abgeschirmte Buchse. Diese ähnelt dem Centronics-Anschluß sehr stark, hat aber 50 statt nur 36 Kontakte. Während die Kontaktleisten für einen 50-poligen Platinenstecker im Elektronikhandel sehr leicht erhältlich sind, dürfte man mit diesem Stecker erheblich mehr Probleme haben.

Außerdem stimmt die auf der Buchse eingeprägte Numerierung der Pins nicht mit der im Handbuch überein: Auf der Buchse sind für die obere Kontaktreihe 1 bis 25 und für die untere 26 bis 50 angegeben. Die Numerierung für die Platinenstecker wechselt jedoch ständig von der Ober- zur Unterseite, so daß oben nur die ungeraden und unten nur gerade Nummern vorkommen.

Die folgenden Angaben verwenden alle die von oben nach unten wechselnde Numerierung der Platinenstecker. Das hat den Vorteil, daß die Adern in einem angepreßten Flachbandkabel wieder genau in dieser Reihenfolge vorliegen: Anschluß 1 ist identisch mit der ersten Ader, Anschluß Nummer 2 mit der zweiten usw. bis hin zur 50. und letzten. Für die Besitzer des CPC 6128 ist das aber nicht weiter tragisch, da sie sich immerhin in soweit an den Nummern auf der Buchse orientieren können, als hier Pin 1 und Pin 50 in beiden Zählarten identisch sind.

Die folgende Grafik stellt eine Aufsicht von hinten auf den Platinenstecker dar. Die Anschlußnummern für den CPC 6128 sind noch einmal in Klammern hinzugefügt.



Zwischen Kontakt 21 und 23 (bzw. zwischen 22 und 24) ist die Platine eingefräst, um einen Verpolungsschutz zu ermöglichen.

Diese Anschlüsse sind wie folgt belegt:

1 ——— Tonsignal (Kanal A+B+C, wie für den eingebauten Lautsprecher)
2 – GND = Masse-Anschluß

Adressbus:

3 ——— Adreßbit 15
4 – Adreßbit 14
5 ——— Adreßbit 13
6 – Adreßbit 12
7 ——— Adreßbit 11
8 – Adreßbit 10
9 ——— Adreßsbit 9
10 – Adreßbit 8
11 ——— Adreßsbit 7
12 – Adreßbit 6
13 ——— Adreßbit 5
14 – Adreßbit 4
15 ——— Adreßbit 3
16 – Adreßbit 2
17 ——— Adreßbit 1
18 – Adreßbit 0

Datenbus:

19 ——— Datenbit 7
20 – Datenbit 6
21 ——— Datenbit 5
22 – Datenbit 4
23 ——— Datenbit 3
24 – Datenbit 2
25 ——— Datenbit 1
26 – Datenbit 0

27 ——— Vcc = +5 Volt Versorgungsspannung

Steuerbus:

28 –	(0) MREQ	Aus	Aus – Ausgang
29 ———	(0) M1	Aus	Ein – Eingang
30 –	(0) RFSH	Aus	oK – open Kollektor
31 ———	(0) IORQ	Aus	
32 –	(0) RD	Aus	
33 ———	(0) WR	Aus	
34 –	(0) HALT	Aus	
35 ———	(0) INT	oK Ein	
36 –	(0) NMI	oK Ein	
37 ———	(0) BUSRQ	oK Ein	
38 –	(0) BUSAK	Aus	
39 ———	(0) WAIT = (1) READY	oK Ein/Aus	
40 –	(0) BUS RESET	oK Ein	
41 ———	(0) RESET	Aus	
42 –	(0) ROMEN	Aus	
43 ———	(1) ROMDIS	oK Ein	Eins-aktiv!
44 –	(0) RAMRD	Aus	
45 ———	(1) RAMDIS	oK Ein	Eins-aktiv!
46 –	(1) CURSOR	Aus	
47 ———	(1) LIGHTPEN	Ein	
48 –	(0) EXPANSION	Ein	(Brücke gegen GND im AMSDOS-Controller)
49 ———	GND = Masse		
50 –	Takt 4 MHz	Aus	

Die Anschlüsse 40 bis 45 bedürfen einer Erläuterung:

40 – BUS RESET (0)

Dieser Eingang sollte von einem Peripheriegerät benutzt werden, um den Computer total zurückzusetzen (Kaltstart). Solange dieser Eingang auf logischem Null-Pegel liegt, wird an Pin 41 ebenfalls ein Reset-Signal ausgegeben.

41 – RESET (0)

Dieser Anschluß dient nur als Ausgang, um die angeschlossene Peripherie Hardware-mäßig zurückzusetzen. Als Software-Reset dient ein OUT &FBFF,xx.

42 – ROMEN (0)

Die ULA erzeugt dieses Signal bei jedem Speicher-Lesezugriff, der nach ihrer Programmierung (ROM-Status) aus einem ROM erfolgen muss. Es kann direkt als Chip-Select-Signal für alle angeschlossenen ROMs verwendet werden. Zusammen mit dem Output-Enable-Eingang eines ROMs (das nur beim selektierten ROM aktiv werden darf) kann das angewählte ROM eingeblendet werden. ROMEN wird aber auch aktiv, wenn die CPU auf das untere ROM mit dem Betriebssystem zugreift.

43 – ROMDIS (1)

Das eingebaute ROM mit Betriebssystem und dem BASIC-Interpreter verfügt über keine eigene Select-Decodierung. Es muss von einem externen ROM (wie AMSDOS) ausgeblendet werden. Auch wenn sich zwei verschiedene ROMs auf derselben Adresse angesprochen fühlen, kann das weiter hinten am Bus angesteckte ROM das vordere mit dieser Leitung ausblenden, wenn eine Daisy-Chain korrekt installiert wurde. Da letzteres aber sehr unwahrscheinlich ist, wird man oft darauf verzichten.

44 – RAMRD (0)

Bei jedem Lesezugriff, der sich laut Programmierung der ULA auf das eingebaute RAM bezieht, wird diese Leitung aktiviert.

45 – RAMDIS (1)

Wird diese Leitung auf logischen Eins-Pegel gelegt, so wird jeglicher Lesezugriff auf das eingebaute RAM unterbunden (ausgenommen der Video-Controller). Dadurch kann man zusätzliche RAM- und ROM-Kontrollogiken installieren, um unabhängig von der Programmierung der ULA auf ein bestimmtes ROM zugreifen zu können. Auf diese Weise ist eventuell eine NMI-Behandlung sinnvoll realisierbar.

Dieser Eingang ist leider nur für Lesezugriffe auf das eingebaute RAM wirksam. Schreibbefehle an das RAM können hiermit nicht unterbunden werden. Damit haben die Hardware-Entwickler praktisch eine selbstgebaute RAM-Erweiterung, die hinten an den Systembus angesteckt werden soll, unmöglich gemacht.

Bei den "Bastelanleitungen" befindet sich ein einfacher Vorschlag, wie die Funktion von RAMDIS auch auf Speicher-Schreibbefehle ausgedehnt werden kann.

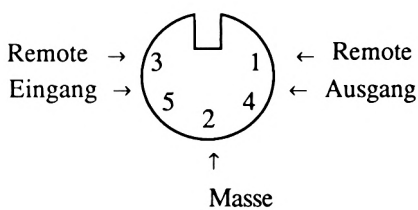
DER ANSCHLUSS FÜR DEN KASSETTENRECORDER

Die CPCs 664 und 6128 werden mit einem eingebauten 3-Zoll-Laufwerk als Massenspeicher verkauft. Trotzdem ist im Betriebssystem-ROM nach wie vor der *Cassette Manager* vorhanden. Er kann mit den RSX-Befehlen |TAPE, |TAPE.IN und |TAPE.OUT aktiviert werden. Das wäre natürlich sinnlos, wenn keine Möglichkeit bestände, auch einen Kassettenrecorder anzuschließen.

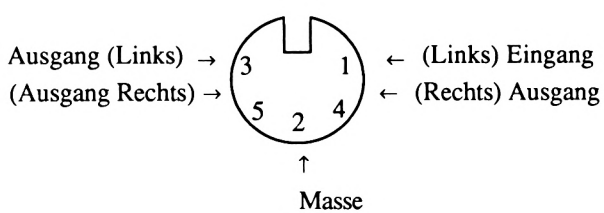
Als Verbindung wurde eine fünfpolige DIN-Buchse verwendet. Leider ist sie nicht normgerecht beschaltet, so daß man um ein selbst gelötetes Adapterkabel nicht herumkommt. Als Kassettenrekorder empfiehlt sich ein Monogerät, das über einen Remote-Eingang verfügt.

Da der Kassetten-Manager alle Dateien in 2 kByte große Blocks zerlegt und diese Stück für Stück abspeichern und laden kann, müßte man ständig selbst die PauseTaste des Rekorders bedienen. Besonders beim Abspeichern kann dabei schon einmal der Anfang eines Blocks verlorengehen. Über den Remote-Eingang kann der CPC 664/6128 aber den Motor des Kassettenrecorders selbst ein- und wieder ausschalten.

Die folgende Grafik zeigt eine Aufsicht auf die Buchse:

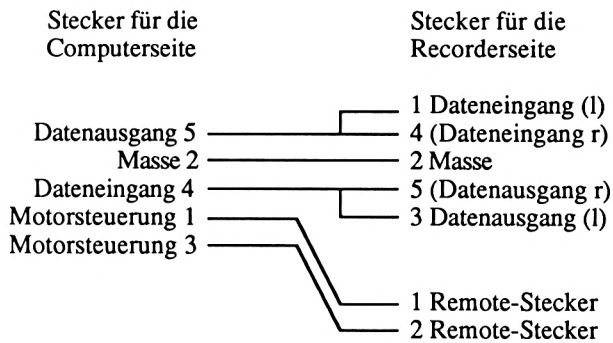


Belegung des DIN-Steckers am Kassettenrecorder:



Ein Mono-Kassettenrecorder verfügt möglicherweise nur über einen dreipoligen DIN-Stecker. Dann fehlen die Pins 4 und 5. Aber auch wenn er eine fünfpolige Buchse hat, sind diese Kontakte möglicherweise nicht angeschlossen. Bei einem Stereo-Recorder sollten auf jeden Fall der linke und rechte Kanal jeweils zusammen angeschlossen werden, damit die Daten nicht nur auf einer (halben) Spur aufgezeichnet werden.

Das "ideale" Adapterkabel sieht also wie folgt aus:



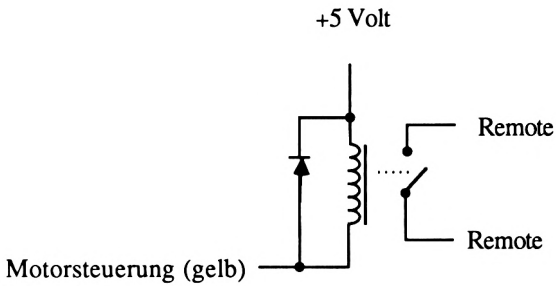
Auch Besitzer des CPC 464 können in die Verlegenheit kommen, von ihrem eingebauten Kassettenrecorder Abstand zu nehmen und einen anderen zu benutzen. Dann nämlich, wenn sie mit höherer Geschwindigkeit Daten aufzeichnen und laden wollen. Von der CPU her sind nämlich bis zu 7000 Baud und mehr möglich, nur die Elektronik des eingebauten Recorders spielt da nicht mehr mit. In diesem Fall ist die Belegung des Kabels interessant, mit dem im Computer-Gehäuse die Rechnerplatine mit der des Kassettenrecorders verbunden ist.

Der Recorder-Anschluß im CPC 464:

Rot – +5 Volt Ausgang (für die Rechnerplatine, vom Ein/Aus-Schalter)
Schwarz – GND – Masse
Weiß – +5 Volt Eingang (vom Monitor)
Blau – Datenausgang (zum Rekorder)
Grün – Dateneingang (vom Rekorder)
Braun – GND – Masse
Grau – Audio – Tonausgang zum Verstärker für den eingebauten Lautsprecher
Gelb – Motorsteuerung

Für die Motorsteuerung wird bei allen CPC-Rechnern ein Relais verwendet. Dieses Relais sitzt beim CPC 464 aber auf der Recorderplatine. Das Verbindungskabel ist also erst die Ansteuerung des Remote-Relais. Hierfür wird auf der Rechnerplatine ein Transistor verwendet, der nach Masse durchschaltet.

Die Spule des Relais muß also an +5 Volt und an das gelbe Kabel angeschlossen werden. Damit beim Ausschalten des Relais-Stromes keine hohen Spannungsspitzen auftreten (wegen der Induktivität der Spule) muß eine Diode in Sperrrichtung parallel geschaltet werden, um den Transistor zu schützen:



Bei den Datenein- und -ausgängen ist ebenfalls eine gewisse Vorsicht angebracht: An dieser Stelle werden noch TTL-Signale (0/+5 Volt) übertragen. Ein einfaches Verbindungskabel zur Stereo-Anlage genügt also nicht.

DER ANSCHLUSS FÜR DAS ZWEITE DISKETTENLAUFWERK

Beim CPC 664 und 6128 ist das erste 3-Zoll-Floppy-Laufwerk bereits in das Grundgerät eingebaut und intern natürlich korrekt angeschlossen. Der Anschluß eines zweiten Laufwerks bereitet keine Schwierigkeiten, sofern man dafür ein fertig konfektioniertes Kabel erwirbt.

Beim CPC 664 ist der Anschluß an der Rückseite wieder nur als Platinenstecker herausgeführt, beim CPC 6128 ist eine Centronics-Buchse verwendet worden.

Diese Buchse hat zwei Nachteile: Zum einen kann sie sehr leicht mit dem Druckeranschluß verwechselt werden, weil es sich in beiden Fällen physikalisch um denselben Anschluß handelt. Zum anderen weicht die Numerierung der Anschlüsse von der im Handbuch (die sich noch auf den Platinenstecker bezieht) so stark ab, daß der erste Versuch, ein Verbindungskabel selbst herzustellen, wahrscheinlich scheitern wird.

Wie beim Druckeranschluß unterscheiden sich der Platinenstecker beim CPC 664 und die Buchse beim CPC 6128 in der Anzahl ihrer Anschlüsse: Die (Centronics-) Buchse hat 36 Kontakte, der Platinenstecker aber nur 34, um handelsübliche Steckkontakte verwenden zu können.

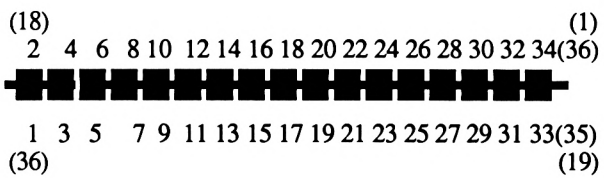
Der Floppy-Anschluß weist im Vergleich zum Drucker-Port aber noch eine Besonderheit auf: Während bei letzterem alle unteren Pins, sofern sie überhaupt angeschlossen sind, auf Masse liegen (als Abschirmung für den Anschluß auf der Oberseite), sind beim Floppy-Anschluß alle oberen Kontakte mit Masse verbunden.

Das hat die Hardware-Designer bei AMSTRAD möglicherweise bewogen, auch die Pin-Numerierung auf den Kopf zu stellen. Während beim Drucker- und Systembus-Anschluß der Kontakt rechts oben (Sicht von hinten) jeweils die Nummer 1 erhielt, ist es hier der Kontakt links unten.

Da beim CPC 6128 die Floppy-Buchse aber in derselben Lage wie der Drukkeranschluß eingebaut ist, kann die Numerierung hier nicht mehr übereinstimmen. Erschwerend kommt noch hinzu, daß beim Platinenstecker die Kontakte wieder alternierend durchnumeriert sind (oben alle geraden, unten alle ungeraden Anschlußnummern), während bei der (Centronics-) Buchse für das Floppy-Laufwerk die Numerierung oben von 1 bis 18 und unten von 19 bis 36 geht.

Im folgenden wird die Numerierung des Platinensteckers beim CPC 664 verwendet, da diese auch mit der Numerierung des Anschlußkabels beim Controller für den CPC 464 übereinstimmt. In der Skizze des Platinensteckers sind die Nummern für den CPC 6128 in Klammern angegeben.

Anschlußbelegung des Floppy-Ports:



Zwischen Kontakt 4 und 6 (3 und 5) befindet sich eine Ausfräsung, um einen Verpolungsschutz realisieren zu können.

Belegung der einzelnen Anschlüsse:

1 ——— (0) READY
2 – GND
3 ——— (0) SIDE 1 SELECT
4 – GND
5 ——— (0) READ DATA
6 – GND
7 ——— (0) WRITE PROTECT
8 – GND
9 ——— (0) TRACK 0
10 – GND
11 ——— (0) WRITE GATE
12 – GND
13 ——— (0) WRITE DATA
14 – GND
15 ——— (0) STEP
16 – GND
17 ——— (0) DIRECTION INWARDS
18 – GND
19 ——— (0) MOTOR ON
20 – GND
21 ——— n.c. (+5 Volt Drive A -> Controller)
22 – GND
23 ——— (0) DRIVE B SELECT
24 – GND
25 ——— n.c. (DRIVE A SELECT)
26 – GND
27 ——— (0) INDEX
28 – GND
29 ——— n.c. (+5 Volt Drive A -> Controller)
30 – GND
31 ——— n.c. (+5 Volt Drive A -> Controller)
32 – GND
33 ——— n.c. (+5 Volt Drive A -> Controller)
34 – GND
35 ——— n.c. (existiert nicht)
36 – n.c. (existiert nicht)

Im großen und ganzen handelt es sich um einen Shugart-Bus. Tatsächlich lassen sich viele 5.25-Zoll-Laufwerke anschließen, wenn man die Beschränkung auf 40 Spuren und eine Seite in Kauf nimmt.

Die mit n.c. (not connected – nicht angeschlossen) gekennzeichneten Leitungen sind beim 464-Controller mit den in Klammern angegebenen Funktionen belegt. Leitung 25 dient als Select-Signal für Laufwerk A, und über die anderen vier Leitungen wird der Controller vom A-Laufwerk mit Strom versorgt.

In den beiden Steckern, die auf das Kabel des 464-Controllers aufgepreßt sind, waren kleine Tricks notwendig, um zwei identische (!) Laufwerke zu Laufwerk A oder B zu machen. Im Stecker für Drive A ist der Pin 23 (Drive B Select) entfernt worden und im Stecker für Laufwerk B die Pins 25 (Drive A Select) und die Pins 21, 29, 31 und 33 (+5-Volt-Versorgung).

Die Schneider-Laufwerke fühlen sich also angesprochen, wenn auf Pin 23 *oder* Pin 25 ein Select-Signal erscheint. Am Stecker liegt es, daß nur Select A oder B bei ihnen ankommt. Außerdem stellen beide 5 Volt zur Versorgung des Controllers bereit, die natürlich nur beim A-Laufwerk benutzt werden dürfen.

Die Verbindungen, die im 464-Aufpreßstecker für Laufwerk B unterbrochen wurden, sind beim CPC 664 und 6128 erst gar nicht angeschlossen.

Wer ein Fremdlaufwerk an den Schneider CPC 464 anschließen will, muß aufpassen, ob er die Leitungen mit +5 Volt von Laufwerk A hier unterbrechen muß. Wer gar zwei Fremdlaufwerke am CPC 464 anschließen will, muß den Controller vom Computer aus mit Strom versorgen. Das geht sogar recht einfach. Im Controller-Gehäuse ist über dem Systembus-Anschluß ein Kondensator eingesetzt, der auf der Platine mit C115 bezeichnet ist. Dieser muß entfernt und durch eine Drahtbrücke ersetzt werden. Dann dürfen Sie aber kein Schneider-Laufwerk mehr als Laufwerk A anschließen, ohne die 4 Leitungen mit der Stromversorgung zu unterbrechen! Ohne Probleme laufen z. B. zwei Hitachi-3-Zoll-Laufwerke am CPC 464.

Die CPC 664- und 6128-Benutzer, die sich das Anschlußkabel für ein zweites Laufwerk selbst basteln wollen, müssen an noch einem Punkt aufpassen. Der Anschluß an den Schneider-Laufwerken ist nicht numeriert. Er entspricht aber direkt dem Ausgang am Computer, nur daß er auf dem Kopf eingebaut ist! Pin 1 ist hier in der rechten oberen Ecke.

Kapitel 3

Das Betriebssystem des Schneider CPC

Die Speicher-Konfiguration im Schneider CPC

Die im Schneider CPC verwendete CPU, ein Z80, kann nominell nur bis zu 64 Kilobyte Speicher adressieren. Trotzdem kommen alle CPCs bereits in ihrer Grundausstattung mit erheblich mehr daher:

1. 64 kByte RAM, die sich über den gesamten Adreßbereich der CPU erstrecken.

2. 16 kByte ROM für das Betriebssystem, die im untersten Adreßviertel dem RAM überlagert werden können.

3. 16 kByte ROM mit dem BASIC-Interpreter, der im obersten Adreßviertel eingeblendet wird.

Alle weiteren ROM-Erweiterungen werden wie der BASIC-Interpreter immer im obersten Adreßviertel eingeblendet. So z.B:

4. 16 kByte ROM mit AMSDOS und den CP/M-Teilroutinen, die beim CPC 664 und 6128 bereits im Grundgerät eingebaut sind. Beim CPC 464 ist dieses ROM im Diskettenkontroller enthalten, den man hinten an den Expansion-Port anstecken kann.

5. Der CPC 6128 enthält darüber hinaus noch weitere 64 kByte RAM, die in acht verschiedenen Kombinationen mit dem normalen RAM gemischt eingeblendet werden können.

Alle sonstigen Erweiterungen, die ein ROM mit zusätzlicher Software enthalten, müssen, wenn sie Gebrauch von den vorhandenen Kernel-Routinen machen wollen, im obersten Adreßviertel von &C000 bis &FFFF eingeblendet werden. Es ist aber möglich, ROMs und RAMs an jeder gewünschten Stelle im gesamten Adreßbereich der CPU einzublenden, weil jeder Lesezugriff der CPU auf eingebaute Speicherbausteine mit den Signalleitungen ROMDIS und RAMDIS am Expansion Port abgeblockt werden kann.

Der Einsatz von zusätzlichen RAM-Bänken wird nur dadurch erschwert, daß sich Schreibzugriffe auf das eingeblendete RAM nicht unterbinden lassen. Hier muß man entweder in Kauf nehmen, daß beim Beschreiben von externem RAM auch die interne RAM-Zelle mit derselben Adresse verändert wird, oder man muß mit dem Eingriff, der bei den Hardware-Basteleien beschieben ist, die Funktion von RAMDIS auch auf Schreibzugriffe ausdehnen. Käufliche Speichererweiterungen umgehen das Problem, indem die Zusatzplatine zwischen der CPU, die dafür aus ihrer Fassung genommen werden muß, und dem Rest der Rechnerplatine untergebracht wird.

Bei den folgenden Betrachtungen tauchen sehr oft die Bezeichnungen "Bank" und "Block" auf. Damit sind die beiden folgenden Sachverhalte gemeint, die leider in der Computer-Literatur noch keine einheitliche Bezeichnung erhalten haben

Bank

Eine "Bank" umfaßt 64 kByte Speicher, was den gesamten Adreßumfang des Z80 darstellt.

Block

Ein "Block" umfaßt alle Speicherzellen, die immer nur gemeinsam ein- oder ausgeblendet werden können. Das sind beim Schneider CPC, wenn man sich an die Vorgaben der Kernel-Routinen hält, immer 16 kByte, also genau ein Viertel des gesamten, von der Z80 adressierbaren Bereichs.

Die Block-Grenzen sind dabei folgende Adressen:

Block 3:——————————→ von &C000 bis &FFFF
Block 2:——————————→ von &8000 bis &BFFF
Block 1:———→ von &4000 bis &7FFF
Block 0: von &0000 bis &3FFF

oder dezimal:

Block 3:——————————→ von 49152 bis 65535
Block 2:——————————→ von 32768 bis 49151
Block 1:———→ von 16384 bis 32767
Block 0: von 00000 bis 16383

DIE RAM-KONFIGURATION

Bei der gesamten Speicherkonfiguration muß man zwischen insgesamt drei Aspekten unterscheiden.

Der erste trifft nur für den CPC 6128 mit seiner zusätzlichen RAM-Bank zu. Normalerweise wird die aktuelle Speicherkonfiguration durch das Gate Array gesteuert. Da der Umschaltmechanismus für zusätzliche 64 kByte RAM im CPC 464 und 664 aber noch nicht vorgesehen war, wurde er auch noch nicht im Gate Array implementiert.

Bei der Entwicklung des CPC 6128 entwickelte man kein neues Gate Array, sondern übertrug den zusätzlichen Verwaltungsaufwand an ein zusätzliches IC, ein PAL.

Diese Feinheiten sind aber für die laufende Software uninteressant, weil zum Programmieren des PAL dieselbe I/O-Adresse wie für die ULA benutzt wird. Hier konnte man eine Adressierungslücke, also eine unbenutzte Adresse, ausnutzen.

ULA und PAL werden durch einen I/O-Zugriff angesprochen, bei dem die Adreßleitung A15 auf Null-Pegel liegt. Beide, ULA und PAL, können nur beschrieben, aber nicht gelesen werden. Daraus ergibt sich die Ansprechadresse:

```
OUT &7FFF, wert
```

oder binär:

```
OUT &X011111??.????????, wert
```

Bei der binär angegebenen Adresse sind die Adreßbits, die mit einem Fragezeichen versehen sind, beliebig. Diese Adreßleitungen werden nicht ausgewertet.

Die zusätzliche Funktionsauswahl zwischen PAL und ULA und innerhalb den verschiedenen Registern der ULA erfolgt über die beiden höchsten Datenbits D6 und D7. Um das PAL zu programmieren, müssen beide auf 1 gesetzt sein:

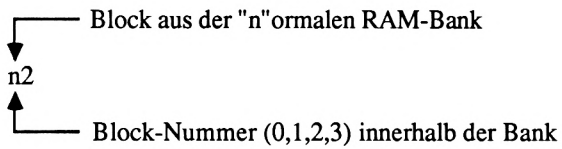
```
OUT &7FFF,&X11??????
```

Um zwischen den acht möglichen RAM-Konfigurationen unterscheiden zu können, genügt es, daß das PAL hierfür nur die Datenbits D0, D1 und D2 aus-

wertet. Die folgende Tabelle zeigt die acht verschiedenen RAM-Konfigurationen, die programmierbar sind:

Datenwort	eingeblendete RAM-Blocks	normalerweise verwendet für:	Bildwiederholspeicher in:
&X11000000	n0 – n1 – n2 – n3	Normalzustand	n3=Screen
&X11000001	n0 – n1 – n2 – z3	CP/M: BIOS, BDOS etc.	n1=Screen
&X11000010	z0 – z1 – z2 – z3	CP/M: TPA	(n1=Screen)
&X11000011	n0 – n3 – n2 – z3	CP/M: n3=Tabellen	(n1=Screen)
&X11000100	n0 – z0 – n2 – n3	Bankmanager	n3=Screen
&X11000101	n0 – z1 – n2 – n3	Bankmanager	n3=Screen
&X11000110	n0 – z2 – n2 – n3	Bankmanager	n3=Screen
&X11000111	n0 – z3 – n2 – n3	Bankmanager	n3=Screen

Die Bezeichnungen für die eingeblendeten RAM-Blocks sind dabei wie folgt zu verstehen:



Die Bildausgabe kann dabei nur aus Blocks der normalen RAM-Bank erfolgen. Der Bildwiederholspeicher kann also vollkommen dem Zugriff der CPU entzogen werden, wenn man ihn durch Programmieren des CRTC in eine normale RAM-Bank legt, die gerade ausgeblendet ist.

Für den normalen Gebrauch sind sicher die Kombinationen 100 bis 111 am interessantesten, mit denen sich alle vier Blocks des zusätzlichen RAMs in Block 1 (&4000 bis &7FFF) des Adreßbereichs der CPU einblenden lassen. Das wird vom Bankmanager benutzt.

Um die verschiedenen RAM-Konfigurationen komfortabel verwalten zu können, wurde im CPC 6128 noch ein zusätzlicher Vektor in die Haupt-Sprungleiste aufgenommen: &BD5B KL RAM SELECT. Dieser Vektor ist wie alle anderen auch im Anhang ausführlich beschrieben.

DIE ROM-KONFIGURATION

Der zweite und dritte, bei allen Schneider CPCs vorhandene Gesichtspunkt der Speicherkonfiguration betrifft die ROMs. Die ROM-Konfiguration unterteilt sich in den ROM-Status, um den sich die ULA kümmert, und die ROM-Auswahl (Selektion). Der ROM-Status legt dabei fest, ob unten das Betriebssystem-ROM und/oder oben irgendein sonstiges ROM eingeblendet werden soll.

ROM-Selektion

Welches ROM aber oben tatsächlich eingeblendet wird, hängt von der ROM-Selektion ab. Hierfür ist kein spezieller Baustein im Computer zuständig. Vielmehr muß sich jede Erweiterung, die am Expansion-Port des CPC angeschlossen wird, um die Selektion selbst kümmern. Dafür ist im Betriebssystem eine spezielle I/O-Adresse vorgesehen:

```
OUT &DFFF, select
```

oder binär:

```
OUT &X11011??.????????, select
```

Auch hier wird wieder nur unvollständig decodiert. Eine Null auf Adreßleitung A13 bei einem OUT-Befehl muß das ROM mit der Nummer, die auf den Datenleitungen ausgegeben wird, selektieren. Alle anderen ROMs müssen abgeschaltet werden. Das Signal ROMEN am Expansion-Port, das direkt von der ULA erzeugt wird, darf nur noch beim selektierten ROM dazu führen, daß dieses eingeblendet wird.

ROM-Status

Der ROM-Status (der sich in den Signalen ROMEN und RAMRD der ULA bzw. am Expansion-Port äußert) wird in Registern der ULA programmiert. Das geschieht mit einem OUT-Befehl auf der ULA-Adresse, bei dem die Datenbits D6 und D7 beide auf 0 gesetzt sind:

```
OUT &7FFF,&X000roumm
```

Die Bits D0 und D1 "mm" bestimmen dabei gleichzeitig den Bildschirmmodus und ein gesetztes Bit D4 "r" setzt einen Monitor-Zeilenzähler zurück, über den die Interrupts gesteuert werden. D4 sollte deshalb immer auf Null gesetzt werden.

Die Datenbits D2 "u" und D3 "o" schließlich bestimmen den ROM-Status: Ein gesetztes Bit "u" blendet das untere ROM mit dem Betriebssystem aus, und ein gesetztes Bit "o" macht dasselbe mit dem oberen ROM. Schreib- und Lesezugriffe in diesem Bereich gehen dann jeweils an die entsprechenden Speicherzellen im RAM.

Ist das Bit D2 "u" beziehungsweise Bit D3 "o" nicht gesetzt, also Null, so wird bei Lesezugriffen das entsprechende ROM aktiviert. Schreibzugriffe gehen auch weiterhin an das eingebaute RAM.

Der zweite Registersatz

Im normalen Betrieb enthält das BC-Register des zweiten Registersatzes des Z80 ständig den passenden Wert, um mit OUT (C),C den aktuellen ROM-Status herzustellen.

Das wird vor allem von der Interrupt-Routine benötigt, die ja zum größten Teil im Betriebssystem, also im unteren ROM, untergebracht ist. 300mal in jeder Sekunde muß deshalb das untere ROM eingeblendet werden, unabhängig davon, welcher ROM-Status gerade durch das Hauptprogramm eingestellt ist.

Nach Abschluß des Interrupts muß dann wieder der alte ROM-Status hergestellt werden. Dies läßt sich mit dem Wert aus dem B'C'-Register besonders schnell erledigen.

Da die Interrupt-Routine diesen Wert im B'C'-Register benötigt und auch die anderen Doppelregister des zweiten Registersatzes verändern kann, darf dieser vom Hauptprogramm normalerweise nicht benutzt werden.

Auch das A'F'-Register wird von der Interrupt-Routine benutzt, um externe Interrupts zu erkennen. Normalerweise muß hier das CY-Flag zurückgesetzt sein.

Eine einfache Möglichkeit, die Zweit-Register doch kurzzeitig zu verwenden, ist diese:

```
; Benutzung des zweiten Registersatzes in einem Assembler-Programm
; ----------------------------------------------------------------
BEISP:  DI          ; den Interrupt verbieten und
        EXX         ; B'C' retten (falls es benutzt wird)
        PUSH BC
;
; Das Mcode-Programm --> Man darf fast keine Betriebssystem-Routine
;                        und keinesfalls einen Restart benutzen!
;
        POP  BC
        EXX         ; B'C' restaurieren
        AND  A      ; CY' zurücksetzen (falls benutzt)
        EX   AF,AF'
        EI          ; einen Interrupt wieder zulassen
```

Ist die Ausführungszeit des Programms von DI bis EI kürzer als 125 Mikrosekunden (Befehls-Ausführungszeiten im Anhang!), so kann man sicher sein, daß kein Interrupt übergangen wurde; wenn nicht, so kann der Zähler des Betriebssystems nun nachgehen, was meist aber nicht weiter schlimm ist.

Restarts

Es ist im allgemeinen viel zu aufwendig, sich ständig selbst mit dem Umschalten der Banks zu beschäftigen. Deshalb hat man bei AMSTRAD die Restart-Befehle des Z80 benutzt, um neue, erweiterte Kommandos bereitzustellen, die dem Programmierer hier die Arbeit abnehmen.

Diese Restarts sind im Anhang im Kapitel "THE LOW KERNEL JUMPBLOCK" genau beschrieben. Die wichtigsten dienen aber dazu, einen Z80-CALL- oder -JUMP-Befehl zu ersetzen.

Der Vorteil vor den normalen Z80- Befehlen ist dabei, daß diese Restarts vor Aufruf eines Unterprogramms ROM-Status und eventuell auch die ROM-Selektion so einstellen, wie es das Unterprogramm benötigt.

Das Unterprogramm kann dann ganz normal mit RET abschließen, kehrt aber nicht direkt zum rufenden Programm zurück. Vielmehr haben die Restart-Routinen hier vorgesorgt, daß vorher die alte ROM-Konfiguration wieder hergestellt wird.

Dadurch kann man Unterprogramme in anderen ROMs aufrufen, ohne sich selbst um die Bankeinstellungen kümmern zu müssen. Speziell die Vektoren im zentralen Sprungbereich, dem "MAIN FIRMWARE JUMPBLOCK", sind alle mit solchen Restarts gebildet.

Diese blenden vor Aufruf einer Firmware-Routine das untere ROM ein und nachher wieder aus, so daß ein Hauptprogramm im RAM (oder in einem oberen ROM) überhaupt nicht bemerkt, daß zwischenzeitlich das Betriebssystem-ROM eingeblendet wurde.

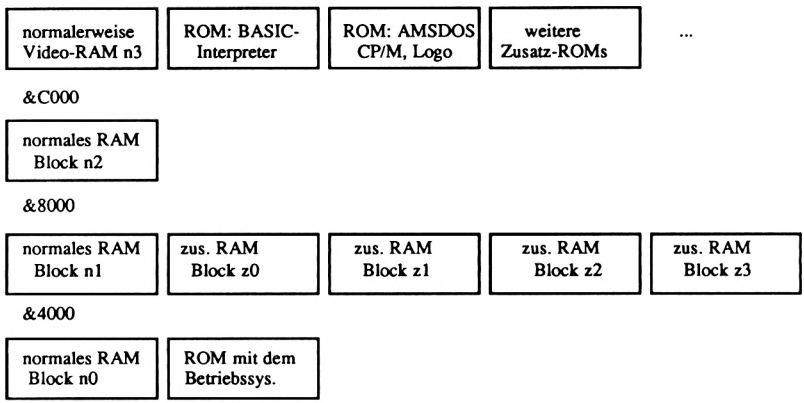
Da die Restarts den normalen Z80-Befehlen CALL und JP so ähnlich wie möglich werden sollten, dürfen sie auch keine Register verändern. Das wird hauptsächlich dadurch erreicht, daß die Restart-Routinen den zweiten Registersatz des Z80 für eigene Berechnungen benutzen.

In dieser Zeit darf aber kein Interrupt eintreten, da dieser ja ebenfalls auf den zweiten Registersatz zugreifen will. Deshalb verbieten alle Restarts kurzzeitig den Interrupt mit DI und lassen ihn nachher mit EI wieder zu.

Alle Restarts schalten den Interrupt wieder ein!

Die folgende Grafik veranschaulicht noch einmal die gesamte Speicherkonfiguration der CPCs, wobei die zusätzlichen RAM-Blocks des CPC 6128 so eingezeichnet sind, wie sie durch den Bankmanager eingeblendet werden.

Speicheraufbau der Schneider CPCs:



EXTERNE HARDWARE

Der Schneider CPC ist dafür ausgelegt, erweitert zu werden, auch wenn man mit der einen oder anderen Schwierigkeit kämpfen muß (bei den RAMs zum Beispiel). Wer eigene Hardware entwickeln und an den CPC anschließen will, muß über einige Informationen verfügen, die sich nicht aus der Anschlußbelegung des Expansion Ports ergeben.

IN/OUT-Adressen

Der erste wichtige Punkt betrifft die frei verfügbaren I/O-Adressen. Diese sind im Handbuch ziemlich unglücklich beschrieben.

Um den Hardware-Aufwand für die Decodierung der einzelnen Adressen so gering wie möglich zu halten, werden beim Schneider CPC wie bei den meisten anderen Z80-Systemen auch die IN- und OUT-Adressen von den Peripheriebausteinen nur sehr unvollständig decodiert.

Der Z80 enthält spezielle Befehle für I/O-Operationen, die sich von den Speicherzugriffen unterscheiden: bei der Programmierung in den verwendeten Befehlen, und nach außen hin in den Signalleitungen IORQ für I/O-Zugriffe und MREQ für Speicher-Schreib- oder -Leseoperationen. Dadurch steht für I/O-Zwecke ein Adreßumfang von normalerweise acht Bit (A0 bis A7) zur Verfügung; bei der im Schneider CPC akzeptierten Beschränkung auf die einfachsten I/O-Befehle sogar der gesamte Adreßbus mit 16 Leitungen.

Deshalb muß man den verschiedenen Peripherie-ICs keine exakt decodierten (Speicher-) Adressen zuweisen, sondern kann die verschiedenen ICs durch jeweils eine einzige Adreßleitung (in Verbindung mit IORQ) direkt anwählen. Das nennt man dann eine unvollständige, Bit-signifikante Adreßdecodierung. Dabei hat man im Schneider CPC die Adreßleitungen "Null-aktiv" definiert: Ein Baustein, wie beispielsweise die PIO oder der Video-Controller, wird durch eine Null auf einer Adreßleitung und an IORQ angesprochen.

Durch die unvollständige Decodierung bleiben für eigene Hardware-Basteleien nur noch sehr wenige Adressen übrig. Die folgende Tabelle zeigt die im Schneider CPC bereits benutzten und die reservierten Adressen. Ganz zum Schluß sind die noch verfügbaren Adressen aufgeführt.

Reservierte und frei verfügbare I/O-Adressen

Baustein	Adresse: (Binär) FEDCBA98 76543210	Bemerkung
Gate Array, PAL	01111111 ???????? ↑	
CRTC Video-Controller	10111100 ???????? 10111101 ???????? 10111110 ???????? 10111111 ???????? ↑	Register adressieren Register beschreiben reserviert Register lesen
ROM Select	11011111 ???????? ↑	
Drucker-Port	11101111 ???????? ↑	
8255 PIO I/O-Schnittstelle	11110100 ???????? 11110101 ???????? 11110110 ???????? 11110111 ???????? ↑	Port A Daten Port B Daten Port C Daten Kontroll-register
Systembus Peripheriegeräte	11111011 01111111 11111011 10111111 11111011 11011111 11111011 11101111 11111011 11110111 11111011 11111011 ↑	Diskettenstation reserviert serielle Schnittstelle ** frei verfügbar ** ** frei verfügbar ** ** frei verfügbar **

Man kann sehr gut sehen, wie mit jeweils einer Adreßleitung (A15 bis A10) ein Baustein angesprochen und mit A8 und A9 manchmal noch eine Zusatzauswahl vorgenommen wird. Die Adreßbits A0 bis A7 sind völlig uninteressant, außer wenn der Erweiterungsbus angesprochen ist (A10 = 0).

Alle Systemerweiterungen, die am Expansion-Port angeschlossen werden und bei den CPCs 664 und 6128 auch der eingebaute Diskettenkontroller, werden zunächst einmal durch eine Null an A10 und IORQ angesprochen. Welches Gerät gemeint ist, wird dann durch die Adreßbits A2 bis A7 festgelegt. Dabei sind A5, 6 und 7 durch AMSTRAD bereits vergeben.

Für eigene Erweiterungen bleiben nur A2, 3 und 4. Man kann diese Bits natürlich auch "vollständig" ausdecodieren und so bis zu 7 verschiedene, eigene Geräte gleichzeitig anschließen. (Die Kombination "111" ist dem Software-Reset vorbehalten; siehe weiter unten.) Das ist aber nicht besonders empfehlenswert, weil man damit den "CPC-Standard" verläßt und mehr Aufwand bei der Decodierung treiben muß.

Wenn man Pech hat, decodieren dann aber zwei Geräte dieselbe Adreßleitung. Deshalb ist es sinnvoll, daß man die Möglichkeit vorsieht, diese zu wechseln. Das kann z.B. mit einem kleinen Schalter auf der Platine geschehen. Dann muß man allerdings auch die Treiber-Software entsprechend gestalten, z.B. durch eine Abfrage bei der Initialisierung oder durch mehrere Versionen mit den drei möglichen Adreßvarianten.

Um verschiedene Funktionen in einer Erweiterung auszulösen, sollte man auf keinen Fall 2 oder alle 3 Bits im Bereich A2 bis A4 benutzen, sondern immer nur zusätzlich zu einer dieser Leitungen die Bits A0, A1, A8 und A9, wie das von AMSTRAD auch vorgesehen ist.

Reset-Decodierung

Bei dieser normalerweise recht unkomplizierten Sache muß man zwischen dem Hardware-Reset, dem Software-Reset und dem Bus-Reset unterscheiden.

Beim Einschalten des Rechners wird ein Hardware-Reset ausgelöst, der sich in einem Null-Pegel der RESET-Leitung äußert. Diese Leitung ist an Pin 41 des Expansion Ports herausgeführt, wo man das Signal zum Zurücksetzen eigener Hardware-Erweiterungen benutzen kann.

Vor allem solche Peripherie, die Interrupts erzeugen kann, muß hiermit ruhiggestellt werden, bis sie explizit wieder aktiviert wird.

Gleichzeitig kann jedes Modul, das hier an den Systembus angeschlossen ist, auch selbst einen Hardware-Reset an Pin 41 auslösen, indem es die Leitung 40 (BUS RESET) am Expansion Port auf logischen Null-Pegel zieht. Das wird beispielsweise in dem Bastelvorschlag für einen Reset-Schalter ausgenutzt.

Nun kennt das Betriebssystem aber auch noch den Software-Reset, der unabhängig von einem Null-Impuls an der Reset-Leitung befolgt werden muß: Immer wenn der Computer im Betrieb initialisiert wird, arbeitet das Betriebssystem einen speziellen OUT-Befehl ab:

OUT &FBFF,&FF = OUT &X 11111011 11111111, &FF

Dieser sollte decodiert und ebenfalls dazu benutzt werden, am Expansion Port angeschlossene Hardware zu re-initialisieren. Bei einem einfachen [CTRL] - [SHIFT] - [ESC] wird kein Hardware-Reset, also kein Null-Pegel auf der Reset-Leitung erzeugt!

Vorausgesetzt, jede am CPC angeschlossene Hardware befolgt vorschriftsmäßig das weiter oben erklärte Adreßkonzept, genügt die Decodierung der folgenden Adreßbits:

&X ?????0?? 111111??

Nur die Null an A10 (Ansprechen des Expansion Ports) und die Einsen an A2 bis A7 sind signifikant.

Für einen korrekten Reset kann man sich auch ausschließlich auf diesen OUT-Befehl verlassen, denn dieser Befehl wird natürlich bei jeder Initialisierung, also auch nach dem Einschalten, ausgegeben. Demgegenüber wird der Hardware-Reset nur beim Einschalten erzeugt (und durch Bus-Reset, was aber bisher nur bei sehr wenigen Erweiterungen ausgenutzt wird).

Trotzdem wird der Software-Reset kaum benutzt. Für seine Decodierung benötigt man nämlich selbst in der einfachsten Form drei ICs der Serie 74LS... mit zusammen 12 TTL-Gattern. Das ist sehr aufwendig. Wie läßt es sich sonst erklären, daß sich das Floppy-Controller-IC (der FDC 765) nicht vollständig zurücksetzen läßt? Hier hätte es einer vollständigen Decodierung der Reset-Bedingung bedurft.

Meist reicht es also aus, nur den Ausgang 41 "RESET" des Expansion Ports zur Initialisierung einer Hardware-Erweiterung auszunutzen. Speziell Interrupt- oder gar NMI-erzeugende Erweiterungen müssen aber auch den Software-Reset testen.

Anschluß eines Zusatz-ROM

Das Bindeglied zwischen externer Hard- und Software ist wohl ein Eprom mit Programmerweiterungen, das am Expansion Port angeschlossen wird. Dafür ist von AMSTRAD als offizieller Weg der Erwerb einer Modulbox proklamiert worden. Man kann sich aber ohne große Schwierigkeiten auch eine eigene Eprom-Platine basteln.

Dazu muß man aber wissen, wie das Betriebssystem des Schneider CPC zusätzliche ROMs handhabt.

Prinzipiell können ROMs im gesamten Adreßbereich eingeblendet werden. Alle Lesezugriffe der CPU auf das eingebaute RAM oder ein ROM können mit den Steuerleitungen ROMDIS und RAMDIS abgeblockt und statt dessen das eigene Eprom eingeblendet werden.

Es ist allerdings für den Normalgebrauch empfehlenswert, sich an die von AMSTRAD vorgezeichneten Regeln zu halten, und zusätzliche ROMs nur im obersten Adreßviertel einzublenden, weil man so in den Genuß der verschiedenen Kernel-Routinen, insbesondere die Restarts kommt.

ROM-Selektion

Jedes ROM erhält eine Selekt-Adresse, die sich normalerweise aus dem Steckplatz in der Modulbox ergibt. Ein ROM wird angewählt (nicht unbedingt auch eingeblendet!), indem auf einer bestimmten Output-Adresse seine Nummer ausgegeben wird. Wird eine andere Nummer ausgegeben, ist das ROM wieder abgeschaltet. Diese Adresse ist &DFFF =&X 110111?? ????????. Eine Null auf A13 zusammen mit IORQ spricht die ROM-Selektion an.

ROM-Status

Ist das ROM selektiert, so muß es eingeblendet werden, wenn ein Speicherzugriff im obersten Adreßviertel erfolgt *und* die ULA die Leitung ROMEN aktiviert. Ob letzteres geschieht, hängt vom ROM-Status ab, der in die ULA programmiert wurde. Bei jedem Lesezugriff auf ein RAM aktiviert diese RAMRD und bei Lesezugriffen auf ROMs ROMEN. Unabhängig vom ROM-Status wird bei Speicher-Schreibbefehlen immer MWE aktiviert.

Fühlt sich ein ROM solchermaßen angesprochen, darf es seine Daten auf dem Datenbus ausgeben, muß aber das eingebaute ROM mit ROMDIS ausblenden. Das BASIC-ROM enthält nämlich keine eigene Selekt-Decodierung, sondern wird immer eingeblendet, wenn es eben nicht ausgeblendet wird.

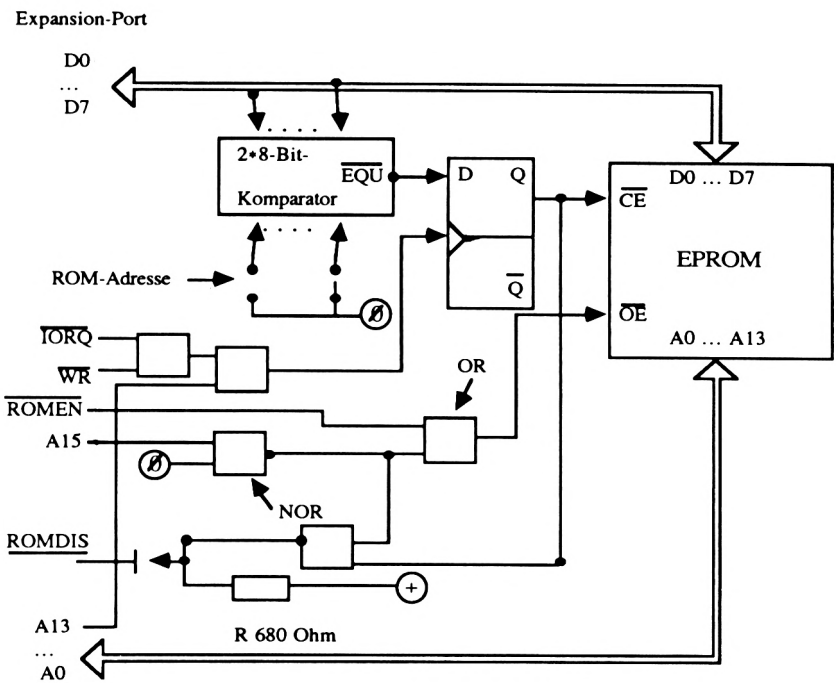
Errata

für das Schneider CPC Systembuch

In Ihrem Buch fehlt auf der Seite 247 die untenstehende Zeichnung. Bitte schneiden Sie die Zeichnung aus, und kleben Sie diese unter die Überschrift.

Wir bitten um Entschuldigung für dieses technisch bedingte Versehen.

Anschluß eines externen ROM-Moduls



Die folgende Zeichnung enthält den Schaltplan für eine solche Eprom-Platine. Für das benötigte Daten-Flip-Flop kann beispielsweise ein IC 74LS74 eingesetzt werden. Jedes andere, Flanken- oder Null-Impuls-getriggerte Flip-Flop tut es aber auch.

Die Decodierung für die Selekt-Adresse ist das umständlichste an der ganzen Schaltung. Die hier gewählte Variante mit einem 2*8-Bit-Komparator ist sicher die universellste, bestimmt aber nicht die billigste. Auch bei den ROM-Adressen kann man sich sicherlich auch auf eine mehr oder weniger unvollständige Adreßdecodierung beschränken. Das ist aber immer mit einem gewissen Risiko behaftet, weil sich Probleme im Zusammenspiel mit eventuellen anderen ROMs ergeben können.

So ist es beispielsweise denkbar, die untersten 4 Datenbits über einen 4zu16-Bit-Decoder auszuwerten. Mit den überzähligen Gattern der Schaltung könnten noch zwei oder drei der oberen Datenbits zu einem CS-Signal für den Decoder selbst zusammengefaßt werden, wodurch die ROM-Adresse auch fast vollständig decodiert werden kann.

Anschluß eines externen ROM-Moduls

Die hier gezeigte Schaltung geht dabei von einer Vereinfachung aus, die immer gegeben sein muß: daß sich nämlich niemals zwei ROMs gleichzeitig angesprochen fühlen.

Daisy Chain

Die Leitung ROMDIS sollte korrekterweise zu einer "Daisy Chain" ausgebaut werden. Dabei geht diese Leitung nicht ununterbrochen vom Expansion-Port am Computer bis zum letzten dort angesteckten Modul durch, sondern wird in jedem Modul unterbrochen. Die Leitung, die hinten in das Modul hineinführt, ist ein Eingang; die Leitung, die vorne wieder herauskommt, ist ein Ausgang.

Fühlt sich ein weiter hinten angestecktes ROM angesprochen, wird es in der Hoffnung, so das ROM des BASIC-Interpreters auszublenden, seinen Ausgang ROMDIS aktivieren. Dieses Signal kommt nun am ROMDIS-Eingang unserer Eprom-Platine an und muß von dieser auf elektronischem Wege auf den eigenen Ausgang durchgeschaltet werden, damit das ROMDIS-Signal auch tatsächlich bis zum Computer gelangt.

Wofür dieser Aufwand? Das wird klar, wenn man den Fall betrachtet, daß das weiter hinten angesteckte ROM dieselbe Selekt-Adresse hat wie das Eprom auf der eigenen Erweiterung, was auch bei nominell unterschiedlichen Adressen wegen einer unvollständigen Adreßdecodierung der Fall sein kann! Dann würden bei einem ROM-Lesezugriff der CPU zwei Bausteine ihre Daten auf den Bus ablegen. Daß die CPU dann kein lauffähiges Programm mehr lesen kann, ist wohl klar. Im Extremfall kann das aber auch bis zur Zerstörung der ROMs gehen.

Um das zu verhindern, muß die eigene Eprom-Erweiterung nicht nur erkennen, ob sie selbst angesprochen ist, sie muß sich auch mit ihrem ROMDIS-Eingang abschalten lassen. Bei dieser Prioritäts-Steuerung mittels ROMDIS erhalten automatisch alle weiter hinten angesteckten Platinen eine höhere Priorität.

Es ist allerdings auch denkbar, daß die Leitung ROMEN für eine Daisy Chain mißbraucht wird. In diesem Fall leitet eine ROM-Platine, die sich angesprochen fühlt, einfach das Signal ROMEN nicht nach hinten weiter. Dadurch erhalten weiter vorne angesteckte Platinen eine höhere Priorität.

Im allgemeinen wird man aber auf solche Spielereien verzichten, da alle angeschlossenen ROMs eigentlich leicht unterscheidbare Selekt-Adressen haben können (252 verschiedene Adressen sind möglich).

Benötigt werden diese Tricks aber, wenn ROMs, von deren Existenz man weiß, ganz bewußt ausgeblendet werden sollen; wenn man beispielsweise das ROM des BASIC-Interpreters komplett ausschalten will, um eine andere Programmiersprache dafür fest zu installieren; oder, um mit einer eigenen ROM-Platine das ROM von AMSDOS zu ersetzen und so ein eigenes DOS (z. B. für zweiseitige 80-Track-Laufwerke) zu installieren.

Die Aufteilung des RAMs im Schneider CPC

Jedes Programm, das im Schneider CPC laufen soll, muß hier auf einige Gegebenheiten Rücksicht nehmen, die vom Betriebssystem vorgezeichnet sind. Ausgenommen davon bleiben wirklich nur Programme, die keine einzige Routine des Betriebssystems benutzen.

Nicht betroffen sind natürlich auch Programme, die in einer höheren Programmiersprache, also Beispielsweise BASIC, geschrieben sind. Hier muß sich der Interpreter bzw. Compiler darum kümmern, daß die Speicheraufteilung eingehalten wird.

Grundsätzlich ist die Verteilung der normalen RAM-Bank (und nur um die dreht es sich hier) so geregelt, daß das RAM von unten her ab Adresse &0000 und von oben her ab Adresse &FFFF für verschieden Aufgaben reserviert wird und in der Mitte ein freier Bereich bleibt, der Memory Pool.

SPEICHERBELEGUNG DURCH DAS BETRIEBSSYSTEM

Wird ein Vordergrundprogramm gestartet (beispielsweise der BASIC-Interpreter, CP/M oder ein Spiel in Maschinensprache), so reserviert sich das Betriebssystem oben und unten einen RAM-Bereich und übergibt dem Programm drei Zeiger, die auf die Unter- und Obergrenze des Memory Pools zeigen:

BC = &B0FF – letztes Byte des Memory Pools (total)
HL = &ABFF– letztes Byte des Memory Pools (Vorschlag)
DE = &0040 – erstes Byte des Memory Pools

Diese Zeiger enthalten dabei immer dieselben Adressen. Der Sinn dieser Zeiger wird später klarer, wenn die Aufteilung, die der BASIC-Interpreter in seinem RAM vornimmt, besprochen wird.

Das RAM bis &003F

In diesem Bereich liegen die Restart-Unterprogramme des Z80. Er wird im Schneider CPC der "LOW KERNEL JUMPBLOCK" genannt, weil fast alle Restarts spezielle Unterprogramme des Kernels aufrufen, die den Befehlssatz des Z80 erweitern. Dieser Bereich ist im RAM und im ROM identisch und darf, bis auf zwei Ausnahmen (External Interrupt und User Restart) nicht verändert werden.

Diese Vektoren sind im Anhang alle genau dokumentiert. Die folgende Grafik bietet aber eine grobe Orientierungshilfe. Zwischen den einzelnen Restarts liegen dabei aber noch weitere Einsprungstellen, die weitere sinnvolle Funktionen bereitstellen:

THE LOW KERNEL JUMP BLOCK

&0040	Beginn des Memory Pools.
&003B	External Interrupt. Einsprungadresse für die Behandlungsroutinen für Interrupts, die von Systemerweiterung am Expansion-Port erzeugt wurden.
&0038 RST 7	Interrupt Entry: Einsprungstelle für den Hardware-Interrupt, der 300mal in jeder Sekunde von der ULA erzeugt wird.
&0030 RST 6	User Restart. Dieser Restart ist für das Vordergrund-Programm frei verfügbar. Die ROM-Version ruft dabei den RAM-Restart auf.
&0028 RST 5	FIRM JUMP: Sprung zu einer Routine mit eingeschaltetem unterem ROM.
&0020 RST 4	RAM LAM: Erweitertes LD A, (HL), bei dem nur aus dem RAM gelesen wird.
&0018 RST 3	FAR CALL: Aufruf eines Unterprogramms in jedem ROM oder RAM.

&0010 RST 2	SIDE CALL: Aufruf einer Routine in einem "benachbarten" ROM. Das ist nur bei sehr umfangreicher ROM-Software nötig.
&0008 RST 1	LOW JUMP: Sprung zu einer Routine im unteren ROM mit wählbarem ROM-Status.
&0000 RST 0	RESET ENTRY: Kaltstart.

Aufteilung des RAMs ab &B100

Oberhalb der Adresse &B100 = 45312 liegen mehrere Bereiche, die sich funktional klar trennen lassen. Zwischen dem CPC 464 und den beiden anderen gibt es hier zum Teil erhebliche Unterschiede. Die folgende grobe Unterteilung trifft aber auf alle drei Schneider Computer zu. Abweichende Werte für den CPC 664 bzw. 6128 sind in Klammern angegeben:

DAS RAM AB &B100

&C000 bis &FFFF	Bildwiederholspeicher, Video-RAM.
&BF00 bis &BFFF	Reservierter Bereich für den Hardware-Stack.
&BE40 bis &BE81	Wird von AMSDOS belegt!
&BDCD bis &BDF3 (&BDF6)	Indirections.
&BB00 bis &BD39 (&BD5D)	MAIN FIRMWARE JUMPBLOCK.
&B921 bis &BAE8 (&BAE3)	In das Ram kopierte Routinen.
&B900 bis &B920	HIGH KERNEL JUMP BLOCK.
&B100 bis &B8FF	Systemspeicher für die Firmware-Routine.
&B0FF	Ende des Memory-Pools.

SPEICHER-AUFTEILUNG DURCH EIN VORDERGRUND-PROGRAMM

Unter einem "Vordergrund-Programm" versteht man das Maschinencode-Programm, das nach der Initialisierung des Rechners durch entsprechende Firmware-Routinen gestartet wird. Das kann beispielsweise der BASIC-Interpreter, CP/M oder ein in Maschinencode geschriebenes Spiel sein.

Wird ein Vordergrund-Programm gestartet, so darf es innerhalb der Grenzen &0040 und B0FF, die ihm vom Betriebssystem in den Registern DE und BC übergeben werden, seinen Speicher organisieren, wie es will. Das Betriebssystem macht aber zusätzlich eine Annahme, die sich in dem in HL übergebenen Wert äußert: Daß das Vordergrund-Programm sich nämlich ab &AC00 einen Bereich für Systemvariablen reservieren wird.

Das trifft für den BASIC-Interpreter zu. Andere Vordergrund-Programme können diesen Wert aber nach Belieben ändern oder ganz ignorieren.

Hintergrund-ROMs

Nun können zusätzlich die sogenannten Hintergrund-ROMs initialisiert werden. Der BASIC-Interpreter macht das immer, wofür er den Vektor "KL ROM WALK" aufruft.

Hintergrund-ROMs enthalten Systemerweiterungen. Sie stellen Routinen bereit, die vom Vordergrund-Programm aus aufgerufen werden können. Ein Beispiel ist das ROM mit dem AMSDOS-Disketten-Betriebssystem.

Man kann davon ausgehen, daß jedes Hintergrund-ROM ebenfalls Systemspeicher im RAM einrichten muß (AMSDOS beispielsweise &500 = 1280 Bytes). Der muß nun vom Memory Pool des Vordergrund-Programms abgezogen werden. Dazu übergibt das Vordergrund-Programm den Hintergrund-ROMs bei deren Initialisierung die Zeiger auf die Unter- und die Oberkante des noch frei verfügbaren Systemspeichers in den Registern DE und HL. Die Systemerweiterung (AMSDOS) reserviert sich dann einfach Speicherplatz, indem sie die Werte in diesen Registern verändert.

Zusätzlich zieht auch noch das Betriebssystem pro Hintergrund-ROM 4 Bytes ab, die für die verkettete Liste aller RSX-Kommandos benötigt werden.

Begnügt sich das Vordergrund-Programm mit den vom Betriebssystem angenommenen Werten, benötigt es also keinen festen Variablenbereich am unteren Ende des Memory Pools (wofür erst DE verändert werden müßte) und benutzt es einen statischen (ortsfesten) Variablenbereich ab &AC00, so braucht

es keine Register zu verändern, bevor es die Hintergrund-ROMs initialisiert. Es kann direkt mit den ihm übergebenen Werten KL ROM WALK aufrufen. Das macht beispielsweise der BASIC-Interpreter.

Um sich unten Speicherplatz zu reservieren, muß das Hintergrund-ROM bei seiner Initialisierung das DE-Register entsprechend erhöhen. Um am oberen Ende des Memory Pools Speicherplatz zu erhalten, muß HL vermindert werden. Diese Ausgaben in den HL- und DE-Doppelregistern können dann direkt als Eingabe für das nächste Hintergrund-ROM benutzt werden, sofern es ein solches noch gibt.

Sonst erhält das Vordergrund-Programm diese beiden veränderten Zeiger zurück und erkennt daran, welcher Bereich ihm noch zur Verfügungb steht.

Dabei hat das Vordergrund-Programm, im Normalfall also der BASIC-Interpreter, vor den Hintergrund-ROMs einen entscheidenden Vorteil: Es bekommt vom Kernel immer eine feste Ober- und Untergrenze übergeben, während Hintergrund-ROMs mit dem zufrieden sein müssen, was ihnen von vorher initialisierten Hintergrund-ROMs und vom Vordergrund-Programm übriggelassen wird.

Hintergrund-ROMs haben keine Möglichkeit, in irgendeiner festen Speicherzelle die Adresse ihres dynamischen RAM-Bereichs zu speichern. Sie könnten im Normalfall gar nicht auf ihr reserviertes RAM zugreifen, weil sie nicht wissen, wo es liegt!

Das ist natürlich ein unmöglicher Zustand. Deshalb wird bei jedem Aufruf einer Routine in einem Hintergrund-ROM über

- &0010 LOW KL SIDE CALL (RST 2)
- &0013 LOW KL SIDE PCHL
- &0018 LOW KL FAR CALL (RST 3)
- &001B LOW KL FAR PCHL und
- &0023 LOW KL FAR ICALL

die angesprungene Routine im IY-Register die Untergrenze des dynamischen RAM-Bereichs über dem Memory Pool übergeben, der für ihr ROM reserviert wurde. Mit Hilfe des IY-Registers kann dann auf den über dem Memory Pool liegenden Systemspeicher zugegriffen werden. Um die Verwaltung des unterhalb vom Memory Pool reservierten Systemspeichers müssen sich die Hintergrund-Routinen aber selbst kümmern, was über Zeiger im oberen Systemspeicher geschehen muß.

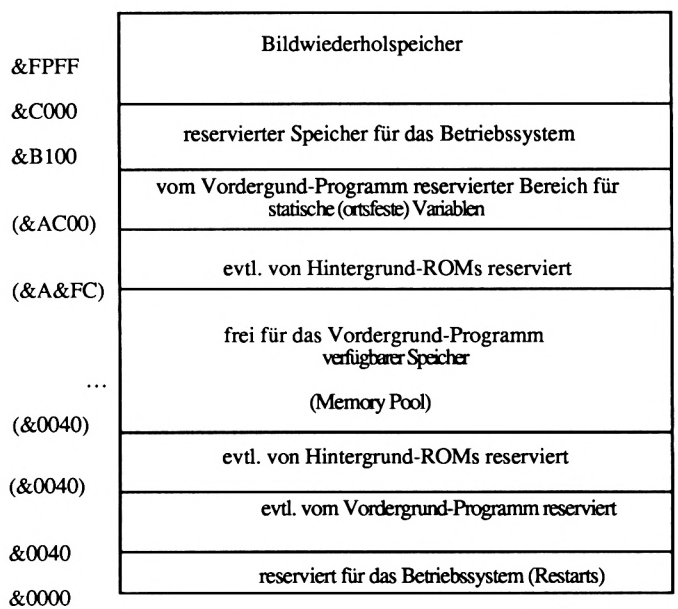
Der Wert, der dabei immer im IY-Register übergeben wird, ist die Adresse des ersten Bytes im oberen Systemspeicher. Dieser Wert ist um 1 höher als der im HL-Register nach der Initialisierung zurückgegebene Wert für die Oberkante des verbliebenen freien Speicherbereichs!

Speicheraufteilung total

Ein Vordergrund-Programm wird also im Normalfall zuerst die untere und obere Grenze erhöhen bzw. vermindern, um sich so einen Variablenbereich mit festen Adressen zu reservieren. Danach werden die Hintergrund-ROMs initialisiert, und dann steht erst der für Anwenderprogramme, Daten o.ä. frei verfügbare Speicherbereich fest.

Daraus ergibt sich die folgende Unterteilung des RAMs, die für alle Vordergrund-Programme zutrifft. Die Adressen der von BASIC zusammen mit AMSDOS reservierten Bereiche sind in Klammern eingetragen.

Speicheraufteilung durch Firmware, Vordergrund-Programm und Hintergrund-ROMs

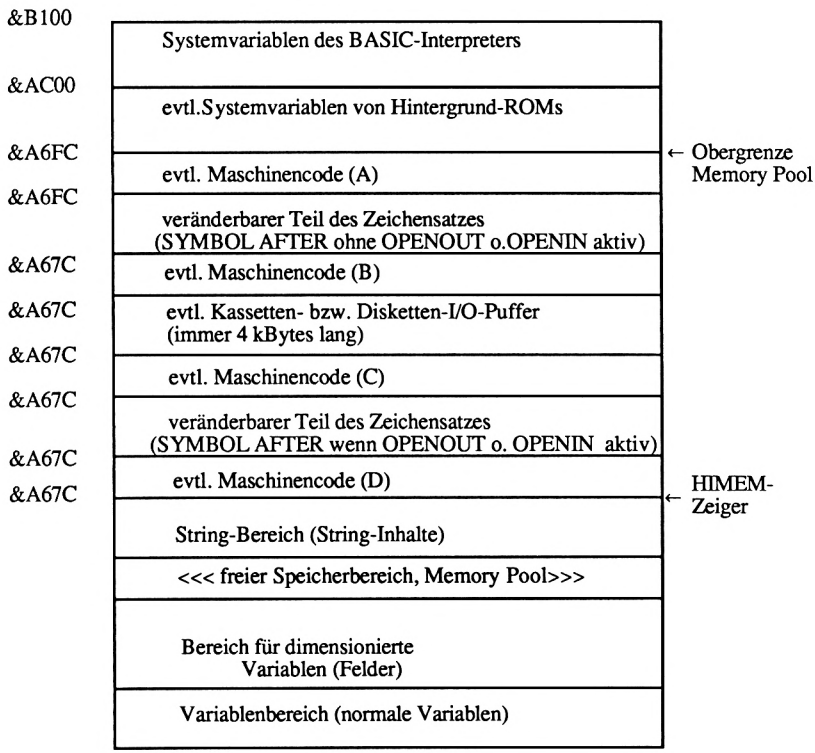


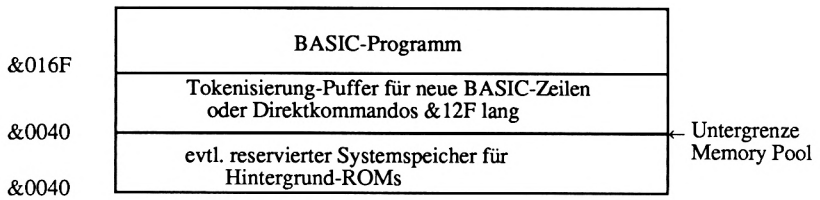
DIE AUFTEILUNG DES RAMS DURCH DEN BASIC-INTERPRETER

Bei Aufruf des BASIC-Interpreters steht ihm als Vordergrund-Programm der RAM-Bereich von &0040 bis &B0FF zur Verfügung. Ab &AC00 wird das RAM für statische Systemvariablen benutzt, so wie es ihm vom Betriebssystem "vorgeschlagen" wird. Abgesehen davon sind aber alle weiteren Daten, die BASIC speichern muß, und dazu gehört beispielsweise auch das BASIC-Programm, das der Anwender eingegeben hat, frei verschieblich.

Das ist nur dadurch möglich, daß in dem statischen Variablenbereich oberhalb von &AC00 Zeiger gespeichert werden, die auf die Grenzen der zu verwaltenden Bereiche zeigen. Insgesamt ergibt sich das folgende Bild. Dabei sind die Adressen wieder für "BASIC plus AMSDOS" angegeben:

Speicheraufteilung durch BASIC





Oberhalb &AC00 liegen die statischen Systemvariablen des BASIC-Interpreters.

Darunter kommt der Bereich, in dem sich Hintergrund-ROMs Speicherplatz für ihre eigenen Systemvariablen reserviert haben. AMSDOS reserviert genau &500 Bytes, so daß, vorausgesetzt, daß sonst keine weiteren ROM-Erweiterungen angeschlossen sind, die Obergrenze des Memory Pools auf &A6FB (42747) sinkt. Prinzipiell ist die Länge dieses Bereichs zunächst einmal unbekannt. Sind aber alle Hintergrund-ROMs initialisiert, verändert sich die Obergrenze des Memory Pools nicht mehr.

Der Bereich darunter bis zu &0040 steht dem BASIC-Interpreter noch zur Verfügung, nachdem er alle Hintergrund-ROMs initialisiert hat. Dabei ist zu bedenken, daß "möglicherweise" auch noch unten Speicherplatz von Hintergrund-ROMs beansprucht wird und sich die Untergrenze des BASIC-Speichers nach oben verschiebt. Bei AMSDOS ist das aber nicht der Fall.

Eine weitere feste Einrichtung ist dann noch der Tokenisierung-Puffer, den BASIC ganz unten anlegt. Jede Eingabe, die man mit dem Zeileneditor macht, muß in die interne Kurzcodierung von BASIC umgewandelt werden. Dabei werden vor allem die Befehlswörter durch ein einziges Kürzel, das sogenannte "Token" ersetzt.

Das hat den Vorteil, daß der von BASIC-Programmen beanspruchte Speicherplatz kürzer wird, man also längere Programme schreiben kann, und daß die Programme erheblich schneller abgearbeitet werden.

In aller Regel liegt dann der Beginn des Programmspeichers bei &016F (muß er aber nicht!). Deshalb existiert oberhalb von &AC00 ein Zeiger auf diese Stelle.

Über dem Programm folgt der Bereich, in dem alle nicht dimensionierten Variablen gespeichert sind. Auch auf den Anfang dieses Bereichs existiert ein Zeiger. Das ist auf jeden Fall nötig, da sich die Lage des Variablenspeichers

mit jeder Änderung im Programmtext verschiebt. Hier werden alle Variablen mit ihrem Namen und dem gespeicherten Wert beigelegt. Die Länge der einzelnen Variableneinträge ändert sich mit der Länge des Namens und ihrem Typ: Integer-Zahlen benötigen nur zwei Bytes, Fließkommazahlen 5 Bytes, und für Strings wird der "String-Descriptor" eingetragen, der drei Bytes beansprucht.

Über den normalen Variablen folgen die dimensionierten Felder. Deren Abspeicherung ist noch etwas komplizierter als bei den einfachen Variablen.

Danach beginnt der Bereich, der auch durch den BASIC-Interpreter noch nicht benutzt ist, der verbliebene Memory Pool. In den wachsen von oben her die Strings hinein. In den Variablenbereichen werden ja nur die Descriptoren, also Längenangabe und Zeiger auf den eigentlichen String-Text abegespeichert.

Im Kapitel über Daten ist die Speicherung von Strings näher erläutert. Der Zeiger auf die Oberkante des String-Bereichs ist die auch für BASIC-Programme zugängliche Systemvariable HIMEN.

Bereich über HIMEM

Über HIMEM gibt es einige Bereiche, die nur bei Bedarf eingerichtet werden:

1. Maschinencode-Programme
2. der Kassetten/Disketten-I/O Puffer und
3. Zeichenmatrizen

Wird BASIC gestartet, so ist noch kein I/O-Puffer reserviert. Maschinencode-Programme über HIMEM gibt es auch noch nicht, wohl werden aber die Definitionen der 16 letzten Zeichen des Zeichengenerators, CHR$ (&F0) bis CHR$ (&FF), aus dem ROM hierhin kopiert.

Wird jetzt ein Programm geladen, so wird der I/O-Puffer unterhalb des Zeichensatzes eröffnet. Nach Abschluß des Ladevorgangs wird er aber normalerweise wieder geschlossen.

Man kann aber auch den veränderlichen Teil des Zeichengenerators vollständig aus dem RAM löschen (SYMBOL AFTER 256). Eröffnet man jetzt eine Datei für Ein- oder Ausgabe, so wird der Puffer wieder eingerichtet. Während der Puffer eröffnet ist, kann man wieder mit SYMBOL AFTER

einen Teil des Zeichengenerators in das RAM kopieren, worauf der I/O-Puffer nicht mehr freigegeben werden kann, weil die veränderbaren Zeichenmatrizen jetzt darunter liegen.

Zusätzlich können noch jederzeit Maschinencode-Programme über HIMEM geladen werden. Es sind also viele Kombinationen denkbar, mit denen oberhalb von HIMEM der Platz verteilt wird.

Mit dem Zeichengenerator hat der BASIC-Interpreter jedoch Probleme. Liegt dieser nicht mehr direkt über HIMEM, weil darunter der I/O-Puffer eröffnet wurde oder HIMEM von Anwender herabgesetzt wurde (um darüber ein Maschinenprogramm zu laden), so kann er die Größe dieses Bereichs nicht mehr ändern. Die folgenden Programme zeigen das:

```
10 OPENOUT "test
20 SYMBOL AFTER 256
RUN(ENTER)
Improper argument in 20
Ready
```

oder

```
10 MEMORY HIMEM -1
20 SYMBOL AFTER 256
RUN (ENTER)
Improper argument in 20
Ready
```

Die Fehlermeldungen treten sogar auf, wenn durch das SYMBOL AFTER-Kommando die Anzahl der veränderlichen Zeichenmatrizen nicht geändert wird!

Es ist deshalb sehr oft ratsam, in einem Programm zunächst den Bereich der änderbaren Zeichenmatrizen zu schließen, dann eventuell Mcode-Programme nachzuladen und erst danach wieder die benötigten Zeichen im RAM anzulegen.

Ein weiterer sinnvoller Trick besteht darin, auch den I/O-Puffer so anzulegen, daß ihn BASIC nicht mehr schließen kann. Wenn BASIC diesen Puffer nämlich eröffnet und schließt, muß es jedesmal eine Garbage Collection durchführen. Dazu dient das folgende Programm:

Die Zeilen 30 und 70 stellen dabei noch so etwas wie einen "Trick im Trick" dar: Wird nämlich unter AMSDOS eine Diskettendatei eröffnet, so kontrolliert AMSDOS sofort, ob im aktuellen Laufwerk auch eine Diskette eingelegt

ist. Dazu muß erst einmal der Laufwerkmotor angeworfen werden (dauert genau eine Sekunde), und dann ist vielleicht keine Diskette drin.

Wie oben bereits erwähnt, genügt es aber, irgend etwas unterhalb des Puffers einzurichten, und sei es auch nur ein einziges Byte:

```
10 CLOSEIN:CLOSEOUT       , Sicher ist sicher
20 SYMBOL AFTER 256       , lösche Bereich der veränderlichen Matrizen
30 | TAPE                 , nicht beim CPC 464 ohne AMSDOS
40 OPENOUT ""             , I/O-Puffer eröffnen
50 MEMORY HIMEM-1         , ein Byte dazwischen reservieren
60 CLOSEOUT                , Ausgabestrom freigeben; Puffer bleibt erhalten!
70 | -DISC                , nicht beim CPC 464 ohne AMSDOS
80 MEMORY HIMEM-XYZ       , nun evtl. eine Maschinencode-Erweiterung
90 LOAD "MCODE",HIMEM+1   ,nachladen und evtl. initialisieren
100 CALL HIMEM +1        ,
110 SYMBOL AFTER 32         , evtl. den gewünschten Teil des Zeichengenerators
wieder in das
                           , RAM kopieren, um ihn verändern zu können.
120 REM Programm
```

All das ist aber beim Kassetten-Manager nicht der Fall, weshalb die Aufgabe, den Puffer anzulegen, von diesem übernommen wird.

BASIC-Zeiger

Um diese dynamischen Bereiche zu verwalten, benötigt der BASIC-Interpreter eine Vielzahl an Zeigern, die auf die verschiedenen Grenzen zeigen. Diese sind (ortsfest) oberhalb von &AC00 angelegt. Bei den CPC 664 und 6128 entsprechen sich dabei die Adressen, leider nicht auch beim CPC 464. In der folgenden Tabelle sind deshalb die Werte für den CPC 464 und die beiden anderen getrennt aufgeführt.

Adressen der BASIC-Zeiger:

CPC 464	664/6128	zeigt auf	(Bemerkung)
&AE7B	&AE5E	HIMEM	(letztes benutzbares Byte)
&AE7D	&AE60	BASIC-RAM-Ende	(letztes Byte des BASIC-Bereichs incl. Zeichensatz, I/Q-Puffer und reservierter Bereiche für Maschinencode)
&AE7F	&AE62	BASIC-RAM-Start	(Anfang des Tokenisierung-Puffers)

CPC 464	664/6128	zeigt auf	(Bemerkung)
&AE81	&AE64	Programmstart	(Zeiger auf Endbyte der "nullten Zeile")
&AE 83	&AE66	Programmende	(normalerweise identisch mit Var. Start)
&AE85	&AE68	Variablenstart	(Anfang der einfachen Variablen)
&AE87	&AE6A	Felderstart	(Anfang der dimensionierten Variablen)
&AE89	&AE6C	Felderende	(Zeiger auf erstes Byte des Memory Pools)
&B08D	&B071	Start der Strings	(Zeiger auf letztes Byte des Memory Pools)
&B08F	&B073	Ende der Strings	(normalerweise identisch mit HIMEM)

Der BASIC-Interpreter

Locomotive BASIC ist zwar ein Interpreter, hat aber trotzdem durch eine geschickte Programmierung precompilierende Eigenschaften. Vor allem werden die beiden wichtigsten Zeitfresser im Programmlauf getuned: zum einen das Suchen von Sprungzielen (bei GOTO und GOSUB) und zum anderen das Suchen von Variablen im Speicher.

Um zu verstehen, wie das bei AMSTRAD erreicht wurde, muß auf den BASIC-Interpreter etwas genauer eingegangen werden.

DER PROGRAMMBEREICH

Zunächst einmal wird jede BASIC-Zeile "tokenisiert". Das heißt, alle Schlüsselwörter des BASIC-Interpreters werden im Eingabetext gesucht und durch ein Ein- oder Zwei-Byte-Kürzel, ein sogenanntes "Token", ersetzt. Außerdem werden auch Zahlen in eine "schnellere" Darstellung umgewandelt.

Die Token aller BASIC-Schlüsselwörter sind in einer Tabelle in Anhang I zusammengestellt.

Das folgende BASIC-Programm druckt immer die erste Zeile des Programms auf dem Bildschirm aus, einmal als Hex-Dump, dann die entsprechenden Zei-

chen, und zum Schluß wird die Zeile auch noch einmal gelistet, so daß man die interne Codierung möglichst gut vergleichen kann:

```
100 REM <-- Platz für die zu untersuchende
110 REM              BASIC-Zeile
120 REM ----------------------------------

140 ZONE 3
150 i=&170
160 l=PEEK(i)+256*PEEK(i+1)
170 FOR j=i TO i+l-1
180   PRINT HEX$(PEEK(j),2),
190 NEXT
200 PRINT
210 FOR j=i TO i+l-1
220   PRINT CHR$(1);CHR$(PEEK(j));
230 NEXT
240 PRINT:PRINT
250 LIST 100
```

Aufbau der BASIC-Zeilen

Zu den Befehlen, aus denen eine Zeile besteht, kommen noch fünf Verwaltungsbytes hinzu.

Jede Zeile startet mit vier Bytes, die immer die gleiche Bedeutung haben und wird durch ein Null-Byte abgeschlossen:

```
ZEILE1: DEFW ZEILE2 - ZEILE1      ; Gesamtlaenge der Zeile
        DEFW 100                  ; Zeilennummer
        DEFM "bla bla"            ; Inhalt der BASIC-Zeile
        DEFB 0                    ; Endmarkierung
ZEILE2: ...
```

Zeilenlänge und -nummer werden dabei in der Z80-typischen Art als "Word" abgelegt: zuerst das niederwertige Byte und dann das höherwertige. Zusätzlich zum Verwaltungsaufwand für jede einzelne Zeile besteht der Programmbereich noch aus drei weiteren Bytes:

```
PROGR:  DEFB 0                    ; Startmarkierung.
;
        DEFM "ZEILE1"
        DEFM "ZEILE2"
        ...
        DEFM "ZEILEn"
;
        DEFW #0000                ; Endemarkierung
;
VARIAB: ...
```

Mit dem Null-Byte am Anfang und dem Null-Word am Ende werden einige Funktionen des Interpreters erleichtert.

Das Null-Word steht nämlich an der Stelle, an der der BASIC-Interpreter die nächste Zeile hinter der letzten erwarten würde. Hier müßte die Länge der nächsten, aber nicht mehr existierenden Zeile stehen (jede Zeile fängt ja mit der Längenangabe an!).

Sucht BASIC eine Zeilennummer, die hinter der letzten Zeile liegen würde (beispielsweise um eine neue Zeile, die man gerade eingegeben hat, "einzufügen"), so muß es im allgemeinen bei der ersten BASIC-Zeile anfangen und sich mit Hilfe der Längenangaben in jeder Zeile zur nächsten weiterhangeln, bis die gewünschte Zeilennummer erreicht ist.

Dabei muß ständig getestet werden, ob nicht vielleicht bereits die letzte Zeile erreicht ist und die neue Zeile nicht "eingefügt" sondern "angehängt" werden muß.

Dazu könnte der BASIC-Interpreter ständig die Adresse der Zeile, die er gerade betrachtet, mit der veränderlichen Programm-Endeadresse vergleichen.

Der Test auf die unmögliche Zeilenlänge Null ist aber viel einfacher.

Wenn man sich den Programmbereich als einen Array vorstellt, in dem das Programm als eine verkettete Liste von VL-Records abgelegt ist, dann sind die Längenangaben in jeder BASIC-Zeile der Kettungs-Pointer, und der Kettungs-Pointer "Null" ist die Markierung für das Kettenende: NIL.

In ähnlicher Weise entspricht das Null-Byte am Programmanfang (normalerweise auf der Adresse &016F) dem Abschluß-Byte einer nicht mehr existierenden, nullten Zeile vor der ersten Zeile des Programms.

Innerhalb einer BASIC-Zeile werden alle Statements durch Doppelpunkte (:) getrennt. Dieser Doppelpunkt wird intern aber nicht mit seinem ASCII-Code, sondern durch das Byte &01 gekennzeichnet.

```
; BASIC-Zeile: 100 :
; -----------  -----
Z100:  DEFW 6      ; Zeilenlaenge
       DEFW 100    ; Zeilennummer
       DEFB 1      ; der Doppelpunkt
       DEFB 0      ; Abschlussmarke
```

Es ist interessant, daß einige Befehle, die eine Statement-Grenze implizieren, intern mit dem Doppelpunkt-Code abgespeichert werden:

```
; BASIC-Zeile:     100 'abc
; -----------      --------
Z100:  DEFW 11     ; Zeilenlaenge
       DEFW 100    ; Zeilennummer
       DEFB 01     ; impliziter Doppelpunkt von '
       DEFB #C0    ; Code für '
       DEFM "abc"  ; abc
       DEFB 0      ; Abschlussmarke


; BASIC-Zeile:     101 IF THEN ELSE
; -----------      ----------------
Z101:  DEFW 11     ; Zeilenlaenge
       DEFW 101    ; Zeilennummer
       DEFB #A1    ; IF
       DEFB " "    ; trennendes Leerzeichen
       DEFB #EB    ; THEN
       DEFB " "    ; trennendes Leerzeichen
       DEFB 01     ; impliziter Doppelpunkt von ELSE
       DEFB #97    ; ELSE
       DEFB 0      ; Abschlussmarke
```

Codierung von Zahlen

Auch Zahlen werden nicht mit ihrer Ziffernfolge gespeichert. Das ist jedem bekannt, der schon einmal versucht hat, die folgenden Zahlen bleibend in einer Programmzeile zu verewigen:

```
100 PRINT .5
LIST 100 [ENTER]
100 PRINT 0.5
Ready
```

und:

```
100 PRINT &X00110011
LIST 100 [ENTER]
100 PRINT &X110011
Ready
```

Man kann eine Zahl zwar in jeder erlaubten Form eingeben, im Listing erscheint sie aber immer in der standardisierten Ausgabeformatierung des BASIC-Interpreters. In der gespeicherten Programmzeile kann also keine Information mehr darüber vorhanden sein, wie die Zahl eingegeben wurde, sondern nur noch eine Information darüber, welcher Zahlenwert und ob die Eingabe dezimal, hexadezimal oder binär erfolgte:

```
; BASIC-Zeile:     PRINT 0,1,2,9
; -----------      -------------
ZEILE: DEFW 14     ; Zeilenlaenge
       DEFW 100    ; Zeilennummer
       DEFB #BF    ; PRINT
       DEFB " "    ; trennendes Leerzeichen
```

```
        DEFB #0E    ; Token für die Zahl 0
        DEFB ","    ; Komma (normaler ASCII-Code)
        DEFB #0F    ; Token für die Zahl 1
        DEFB ","    ; Komma
        DEFB #10    ; Token für die Zahl 2
        DEFB ","    ; Komma
        DEFB #17    ; Token für die Zahl 9
        DEFB 0      ; Endemarke
```

Die ganzen Zahlen 0 bis 9, die in dezimaler Schreibweise eingegeben werden, werden durch die Token #0E bis #17 gespeichert. Dadurch entsteht zwar kein Platzgewinn, sie sind so aber leichter von den Zeichen 0 bis 9, die beispielsweise in Strings auftreten können, zu unterscheiden:

```
; BASIC-Zeile:   100 PRINT "0123"
; -----------    ----------------
ZEILE:  DEFW 13         ; Zeilenlaenge
        DEFW 100        ; Zeilennummer
        DEFB #BF        ; PRINT
        DEFB " "        ; trennendes Leerzeichen
        DEFM ""0123""   ; der String wird in der
;                       ; ASCII-Codierung gespeichert
        DEFB 0          ; Endmarke
```

Bei allen anderen Zahlenangaben wird dem Zahlenwert ein Token vorangestellt, das Rückschlüsse darauf zuläßt, wie die Zahl eingegeben wurde (dez/hex/bin) und wie groß sie ist bzw. wie sie codiert ist:

```
; BASIC-Zeile:   100 PRINT 55,300,1.5,&FF,&X111
; -----------    ------------------------------
ZEILE: DEFW 28              ; Zeilenlaenge
       DEFW 100             ; Zeilennummer
       DEFB #BF             ; PRINT
       DEFB " "             ; trennendes Leerzeichen
       DEFB #19             ; TOKEN: Dezimalzahl 10 <= n <= 255
;                           ;        folgt (Byte)
       DEFB 55              ;        Zahlenwert
       DEFB ","             ; Komma
       DEFB #1A             ; TOKEN: Dezimalzahl 256<= n <=
;                           ;        32767 folgt (Integer)
       DEFW 300             ;        Zahlenwert
       DEFB ","             ; Komma
       DEFB #1F             ; TOKEN: Allgemeine Dezimalzahl
;                           ; folgt
       DEFB 0,0,0,#40,#81   ;        Zahlenwert in interner
;                           ;        Fliesskomma-Darstellung
       DEFB ","             ; Komma
       DEFB #1C             ; TOKEN: Integer-Zahl in HEX-
;                           ;        Darstellung folgt
       DEFW #00FF           ;        Zahlenwert
       DEFB ","             ; Komma
```

```
        DEFB #1B            ; TOKEN: Integer-Zahl in BIN-
;                           ;        Darstellung folgt
        DEFW #0007          ;        Zahlenwert
        DEFB 0              ; Endmarke
```

Bleibt nur die Frage, wie es mit negativen Zahlen aussieht:

```
; BASIC-Zeile:    100 PRINT -1,-55,-300,-1.5,-&FF,-&X111
; -----------    ------------------------------------
```

In allen Fällen wird das Vorzeichen nicht in die Zahl hineingezogen (wie es zumindest bei der Fließkomma-Darstellung leicht möglich wäre), sondern zusätzlich mit dem Token &F5 für "minus" dargestellt. Das Token &F5 steht dabei vor dem Zahlen-Token:

```
; BASIC-Zeile:  100 PRINT -1.5
; ------------  --------------
ZEILE: DEFW 14              ; Zeilenlaenge
       DEFW 100             ; Zeilennummer
       DEFB #BF             ; PRINT
       DEFB " "             ; Leerzeichen
       DEFB #F5             ; TOKEN für -
       DEFB #1F             ; TOKEN für Fliesskomma-Zahl
       DEFB 0,0,0,#40,#81   ;       Zahlenwert
       DEFB 0               ; Endemarke
```

Außer den Befehlswörtern (Commands) wie PRINT werden auch alle anderen reservierten Wörter tokenisiert. Diese sind dabei, wie man an der Tabelle im Anhang sehen kann, in mehrere Gruppen untergliedert:

```
    #80 bis     #E1: Commands
    #E3 bis     #FE: Operatoren, sonstige Trennwörter
#FF #00 bis #FF #1D: Funktionen mit einem Argument
#FF #40 bis #FF #49: Funktionen ohne Argument
#FF #71 bis #FF #7F: Funktionen mit mehreren Argumenten
```

Sprungadressen

Etwas komplexer wird die Codierung von BASIC-Zeilen, sobald Sprungziele im Programmtext auftauchen. Die hier notwendigen Zahleneingaben (Zeilennummern) werden nicht wie normale Zahlen abgespeichert, was man beispielsweise daran erkennt, daß man Sprungziele nicht in hexadezimaler oder binärer Form eingeben kann:

```
100 GOTO &64
RUN [ENTER]
Syntax error in 100
100 GOTO &64
```

Geben Sie in das weiter oben abgedruckte Testprogramm die folgende Zeile 100 ein, und starten Sie das Programm zunächst mit RUN 110:

```
100 GOTO 110
```

Das führt zu folgendem Ausdruck:

```
0A 00 64 00 A0 20 1E 6E 00 00
```

Die Bytes &0A und &00 bilden zusammen die Zeilenlänge, &64 und &00 die Zeilennummer. Danach folgt &A0, das Token für GOTO, danach &20, also ein Space " ". Der Code &1E muß wohl das Token für eine Zeilennummer sein, weil nicht eins der bisher bekannten Zahlen-Token (&19 oder &1A usw.) verwendet wurde. Die beiden verbleibenden Bytes vor dem letzten Null-Byte (Schlußmarke) müssen wohl die Zeilennummer sein, &006E ist nämlich dezimal 110.

In die in diesem Kapitel bisher verwendete Assembler-Darstellung umgesetzt, sieht das so aus:

```
; BASIC-Zeile:     100 GOTO 110
; ------------     ------------
ZEILE: DEFW 10      ; Zeilenlaenge
       DEFW 100     ; Zeilennummer
       DEFB #A0     ; GOTO
       DEFB " "     ; trennendes Leerzeichen
       DEFB #1E     ; TOKEN für Zeilennummer
       DEFW #006E   ;       Zeilennummer
       DEFB 0       ; Abschlussmarke
```

Worin besteht nun die "precompilierende" Eigenschaft des Locomotive BASIC-Interpreters?

Wie man an diesem Beispiel sieht, wurde in den tokenisierten Programmtext *nicht* die Adresse der Zeile 110, sondern ganz normal die Zeilennummer des Sprungziels eingetragen. Da das Testprogramm mit RUN 110 gestartet wurde, kann es eigentlich nur daran liegen, daß diese Zeile noch nicht abgearbeitet wurde.

Das ist auch sofort gezeigt. Startet man dieses Programm mit RUN 100, so erhält man ein ganz anderes Bild:

```
0A 00 64 00 A0 20 1D 79 01 00
```

Innerhalb des tokenisierten Programms im Speicher kam es zu Änderungen. Statt des Tokens &1E und der Zeilennummer 110 findet sich hier ein anderes Token &1D und ganz offensichtlich die Zeilenadresse:

```
; BASIC-Zeile:  100 GOTO 110
; -----------   ------------
#0170 ZEILE1: DEFW 10      ; Zeilenlaenge
#0172         DEFW 100     ; Zeilennummer
#0174         DEFB #A0     ; GOTO
#0175         DEFB " "     ; Leerzeichen
#0176         DEFB #1D     ; TOKEN für Zeilenadresse
#0177         DEFW #0179   ;       Zeilenadresse
#0179         DEFB 0       ; Endemarke
#017A ZEILE2: ...
```

Um das zu verdeutlichen, sind in diesem Listing auch noch die Adressen der einzelnen Bytes bzw. Words vorangestellt. Der BASIC-Programmbereich beginnt ja auf Adresse &016F mit einem Null-Byte, und die erste Zeile startet bei &0170. Dieses Wissen wurde auch für das Testprogramm benutzt.

Der Wert &0179 ist die Adresse des Abschluß-Bytes der Zeile vor der anzuspringenden Zeile. Um das zu verstehen, muß man sich noch einmal vor Augen halten, wie der BASIC-Interpreter ein Programm abarbeitet und Sprünge und Unterprogramm-Aufrufe realisiert.

Dazu wird ein Zeiger benutzt, der immer auf das gerade bearbeitete Element im Programmspeicher zeigt. Ein Sprung wird dadurch erreicht, daß dieser Zeiger auf das Sprungziel verstellt wird oder, anders ausgedrückt, mit der Zieladresse geladen wird.

Ist ein Befehl fertig abgearbeitet, so muß der Zeiger immer hinter den gerade bearbeiteten Befehl zeigen (außer, es handelte sich um einen Sprung), auf den trennenden Doppelpunkt (der, wie weiter oben gesagt, sogar vor ' und ELSE eingefügt wird) oder auf das Zeilenende, auf das Null-Byte.

Deshalb muß auch nach dem Sprung der Zeiger auf dem Null-Byte der Zeile davor stehen. Da man auch zur ersten Zeile springen kann, muß also auch vor der ersten Zeile ein Null-Byte stehen! Das ist (u.a.) der Sinn des ersten Verwaltungs-Bytes im Programmspeicher.

Hat BASIC also einen Befehl abgearbeitet, so testet es, ob der Zeiger jetzt auf einen Doppelpunkt (intern codiert durch &01) oder ein Zeilenende &00 zeigt. Bei einem Doppelpunkt muß der Zeiger nur um eins erhöht werden, um mit dem nächsten Befehl weitermachen zu können. Bei einem Zeilenende müssen zunächst noch Zeilenlänge und -nummer überlesen werden.

Bei Unterprogramm-Aufrufen muß natürlich zusätzlich der alte Wert des Programmzeigers auf einen Stapel gerettet werden. Das wird aber im Abschnitt über den BASIC-Stack noch weiter erklärt.

LIST

Diese zwei Methoden, das Sprungziel innerhalb des tokenisierten Textes zu speichern, führt zu einigen Besonderheiten, die es für den BASIC-Interpreter zu berücksichtigen gilt.

Wenn eine Zeile mit einem Sprungbefehl gelistet wird, müssen zwei Fälle unterschieden werden. Entweder, die Zeile wurde noch nicht abgearbeitet, und die Zeilennummer des Sprungziels steht noch an dieser Stelle im Listing, oder die Zeile wurde schon abgearbeitet. Dann steht hier nur noch die Adresse der Zeile. Für die interessiert sich der Programmierer aber nicht. Auch in diesem Fall soll im Listing die Zeilennummer erscheinen. Diese muß dann erst bestimmt werden. Das geht aber noch recht einfach, weil man jetzt die Adresse der Zeile hat und in der Zielzeile nachschauen kann, welche Nummer sie hat.

EDIT

Werden im Programm Änderungen vorgenommen (Zeilen einfügen, löschen oder ändern), so verschieben sich alle BASIC-Zeilen ab der Stelle, an der die Änderungen vorgenommen werden. Danach zielen also Sprungadressen daneben.

Deshalb muß vor jeder Änderung im Programmtext das gesamte Programm durchgegangen werden, um alle Zeilenadressen wieder durch Zeilennummern zu ersetzen. Das führt dazu, daß der BASIC-Interpreter bei längeren Programmen erst eine kurze Pause einlegt, bevor er eine neu eingegebene Zeile übernimmt.

RENUM

Die Tatsache, daß der BASIC-Interpreter während der Programmbearbeitung Zeilennummern durch deren Adressen ersetzt, hat noch einen unerwarteten Nebeneffekt: Hiermit läßt sich ein Programm ganz bequem umnumerieren.

Das Problem dabei ist nicht so sehr, alle betroffenen Zeilen zu finden und ihre Zeilennummer zu ändern. Vielmehr ist es recht aufwendig, die Zeilennummern in allen Sprüngen oder sonstigen Bezügen (GOTO, GOSUB, RESTORE, RESUME, ON...GOSUB usw.) entsprechend zu ändern.

Dazu müssen alle solche Referenzen gefunden und durch die zugehörigen Zeilenadressen ersetzt werden. Danach kann man die Zeilen umnumerieren.

Da die Zeilennummern in den Sprungbefehlen u. ä. nachher beim Listen aus den Zielzeilen selbst bestimmt werden, erübrigt sich ein weiterer Durchgang für alle GOTOs, RESTOREs usw. Die Zeilenadressen haben sich ja nicht mehr geändert.

Variablen

Der zweite Punkt, an dem der BASIC-Interpreter des Schneider CPC beschleunigt wurde, ist der Zugriff auf Variablen. Auch hier muß normalerweise bei jedem Variablenzugriff der gesamte Variablenbereich von vorn an durchsucht werden, bis der angegebene Name gefunden ist. Erst dann kann der Wert gelesen oder auch verändert werden.

Wie bei den Zeilenreferenzen wird auch bei Variablen während des Programmlaufs die tatsächliche Adresse in den tokenisierten Programmtext eingefügt. Hierbei wird allerdings nicht der Name durch die Adresse ersetzt. BASIC reserviert vielmehr von vornherein zwei Bytes im Programmtext. Die folgende BASIC-Zeile wurde noch nicht abgearbeitet:

```
; BASIC-Zeile:     100 var=0
; ------------     ---------
ZEILE: DEFW 13               ; Zeilenlaenge
       DEFW 100              ; Zeilennummer
       DEFB #0D              ; TOKEN für Variable
       DEFW #0000            ;       Platz für Adresse
       DEFB "v","a","r"+#80  ;       Name, letztes Zeichen hat
;                            ;       Bit 7 gesetzt.
       DEFB #EF              ; TOKEN für =
       DEFB #0E              ; TOKEN für 0
       DEFB 0                ; Endekennung
```

Eine Variable in einem BASIC-Befehl wird zunächst durch das Token &0D eingeleitet. Es gibt aber auch noch andere, wie wir gleich sehen werden. Danach kommt die Adresse, die in diesem Fall noch nicht bestimmt ist, und dann der Name mit normalen ASCII-Zeichen. Nur im letzten Buchstaben wird das siebte Bit gesetzt (+&80).

In BASIC ist es möglich, Variablen von drei verschiedenen Typen einzurichten: Integer, Real und String. Neben der globalen Definition der Typ-Zugehörigkeit (Default: A bis Z sind Real, sonst mit DEFINT, DEFREAL und DEFSTR) können die Variablen auch mit einem sogenannten Postfix dem einen oder anderen Typ zugeordnet werden. In diesem Fall erhalten die Variablen andere Token. Die folgende BASIC-Zeile wurde noch nicht abgearbeitet (ergäbe sowieso nur eine Fehlermeldung).

```
; BASIC-Zeile:   100 a!=a%+a$
; -----------   ------------
ZEILE: DEFW 19       ; Zeilenlaenge
       DEFW 100      ; Zeilennummer
       DEFB #04      ; TOKEN für Real-Variable mit !
       DEFW #0000    ;       Platz für Adresse
       DEFB "a"+#80  ;       Variablenname ohne !
       DEFB #EF      ; TOKEN für =
       DEFB #02      ; TOKEN für Integer-Variable mit %
       DEFW #0000    ;       Platz für Adresse
       DEFB "a"+#80  ;       Variablenname ohne %
       DEFB #F4      ; TOKEN für +
       DEFB #03      ; TOKEN für String-Variable mit $
       DEFW #0000    ;       Platz für Adresse
       DEFB "a"+#80  ;       Variablenname ohne $
       DEFB 0        ; Endkennung
```

Sobald ein Befehl einmal abgearbeitet wurde, werden in den Null-Words die Adressen relativ zum Start des Variablenbereichs eingetragen. Das Token wird bei Variablen mit Postfix nicht geändert:

```
; BASIC-Zeile:   100 a!=a%+LEN(a$)
; ------------   -----------------
ZEILE: DEFW 23       ; Zeilenlaenge
       DEFW 100      ; Zeilennummer
       DEFB #04      ; TOKEN für Realvariable mit Postfix !
       DEFW #0005    ;       Adresse
       DEFB "a"+#80  ;       Name
       DEFB #EF      ; TOKEN für =
       DEFB #02      ; TOKEN für Integer-Variable mit
;                    ; Postfix %
       DEFW #0033    ;       Adresse
       DEFB "a"+#80  ;       Name
       DEFB #F4      ; TOKEN für +
       DEFB #FF      ; TOKEN für Funktion:
       DEFB #0E      ;       LEN
       DEFB "("      ;
       DEFB #03      ; TOKEN für String-Variable mit
;                    ; Postfix "$"
       DEFW #0039    ;       Adresse des Descriptors
       DEFB "A"+#80  ;       Name
       DEFB ")"      ;
       DEFB 0        ; Endemarke
```

Wenn übrigens die Variablen a% und a$ im obigen Beispiel noch nicht definiert sind, so werden sie durch diesen Befehl nicht eingerichtet. Dann wird nur bei a! eine Adresse eingetragen, bei den beiden Wert-liefernden Variablen noch nicht.

Wie sieht es nun aber bei Variablen ohne Typangabe aus? Vor der folgenden, untersuchten Zeile wurden diese Deklarationen vorgenommen:

```
DEFINT  a:a=0
DEFREAL b:b=0
DEFSTR  c:c=""

; BASIC-Zeile:   100 a=b+LEN(c)
; -----------   --------------
ZEILE: DEFW 23        ; Zeilenlaenge
       DEFW 100       ; Zeilennummer
       DEFB #0B       ; TOKEN für bekannte Integervariable ohne
;                     ; Postfix
       DEFW #0033     ;       Adresse
       DEFB "a"+#80   ;       Name
       DEFB #EF       ; TOKEN für =
       DEFB #0D       ; TOKEN für bekannte Real-Variable ohne
;                     ; Postfix
       DEFW #0040     ;       Adresse
       DEFB "b"+#80   ;       Name
       DEFB #F4       ; TOKEN für +
       DEFB #FF       ; TOKEN für Funktion:
       DEFB #0E       ;       LEN
       DEFB "("       ;
       DEFB #0C       ; TOKEN für bekannte String-Variable ohne
;                     ; Postfix
       DEFW #0049     ;       Adresse des Descriptors
       DEFB "c"+#80   ;       Name
       DEFB ")"       ;
       DEFB 0         ; Endemarke
```

Solange eine Variablenreferenz ohne den Typ-kennzeichnenden Postfix noch nicht abgearbeitet wurde, wird sie mit dem Token &0D gekennzeichnet. Ist die Referenz einmal ausgewertet worden und die Adresse der Variablen in der Programmzeile gespeichert, so wird das Token, je nach Variablentyp, durch &0B, &0C oder &0D ersetzt.

Wie man sieht, kommt dem Token &0D eine Doppelbedeutung zu: Ist noch kein Zeiger in den Programmtext eingetragen, bedeutet &0D: unbekannter Typ. Ist aber ein Zeiger eingetragen, was daran erkennbar ist, daß die Adresse nicht mehr Null ist, so bedeutet &0D: Real-Variable ohne Typangabe.

Insgesamt gibt es also sieben Fälle, die bei den Variablen-Token vom BASIC-Interpreter unterschieden werden müssen:

&02	– Integer	mit % und Adresse evtl. bestimmt
&03	– String	mit $ und Adresse evtl. bestimmt
&04	– Real	mit ! und Adresse evtl. bestimmt

&0B – Integer ohne % und Adresse bereits bestimmt
&0C – String ohne $ und Adresse bereits bestimmt
&0D – Real ohne ! und Adresse bereits bestimmt

&0D – unbekannter Typ, Adresse noch nicht bestimmt: &0000

DER VARIABLENBEREICH

Auch wenn nun der Zugriff des BASIC-Interpreters auf die Variablen jetzt so halbwegs klar ist, ist die Speicherung der Variablen selbst noch nicht geklärt. Hier sind die Entwickler bei AMSTRAD mit dem Wunsch, einen schnellen Interpreter zu liefern, möglicherweise etwas übers Ziel hinausgeschossen.

Zunächst einmal müssen drei verschiedene Bereiche unterschieden werden:

1. Bereich für undimensionierte Variablen
2. Bereich für dimensionierte Variablen
3. Bereich für die String-Texte

Der normale Variablenbereich schließt sich direkt an den Programmspeicher an. Darüber liegen die Arrays und darüber wieder ist der (noch) freie Memory Pool. In diesen wachsen von oben, von HIMEM her die String-Texte hinein. Dieser Bereich ist im Kapitel Datenspeicherung genauer erläutert worden.

In den beiden Variablenbereichen werden Integer- und Realvariablen und String-Descriptoren so eingetragen, wie sie im Laufe des Programms anfallen.

Einfache Variablen

In diesem Bereich hat jeder Eintrag die folgende Form:

```
VAR:  DEFW  Hangel-Pointer
      DEFM  Name der Variablen; letztes Zeichen hat Bit 7 gesetzt (+&80)
      DEFB  Variablentyp: 1, 2 oder 4
      DEFS  2, 3 oder 5 Bytes mit dem Variablenwert
```

Der Variablentyp ist so gewählt, daß er immer um genau eins kleiner ist als die Anzahl Bytes, die für die Speicherung eines Wertes dieses Typs benötigt werden.

1. Integer-Variablen haben den Typ 1 und benötigen zwei Bytes
2. Real-Variablen haben den Typ 4 und benötigen fünf Bytes
3. String-Variablen haben den Typ 2 und benötigen drei Bytes

Für Strings wird, wie im Kapitel über Datenspeicherung erklärt, nur ein Descriptor eingetragen, der immer drei Bytes lang ist. Das ist nötig, weil im Variablenbereich feste Längenverhältnisse herrschen müssen, um im Programmtext precompilierenderweise die Variablenadressen eintragen zu können.

Die Hangel-Pointer deuten darauf hin, daß im Variablenbereich verkettete Listen installiert sind, und zwar nicht nur eine, sondern gleich 26 Stück! Für jeden Anfangsbuchstaben von A bis Z existiert eine Liste. Die Pointer zeigen immer auf die nächste Variable mit demselben Anfangsbuchstaben, wobei auch hier die Adressenangaben (wie im Programmtext) relativ zum Start des Variablenbereichs gemacht werden.

Der nachfolgende kommentierte Assemblerauszug stellt den Variablenbereich dar, wie er vom folgenden Programm hinterlassen wurde. (Das Programm diente dabei dazu, das Grundgerüst des eignen Variablenbereichs in eine Textdatei zu schreiben):

Programm:

```
10 str%=9
15 e$=""
20 IF str%=9 THEN OPENOUT"datei
30 anf=&ae68:ende=&ae6A    ' Werte fuer CPC 664/6128
40 DEF FNdp(i)=PEEK(i)+256*PEEK(i+1)
50 FOR a=FNdp(anf)TO FNdp(ende)-1
60 e$=HEX$(PEEK(a),2)+" "
70 IF (PEEK(a)AND &7F) > 31 THEN e$=e$+CHR$(PEEK(a)AND &7F)
80 PRINT#str%, e$
90 NEXT
100 CLOSEOUT
```

Variablenbereich:

```
DEFW #0000                ; Pointer: NIL
DEFB "S","T","R"+#80      ; Name
DEFB #01                  ; Typ: Integer
DEFW #0009                ; Inhalt

DEFW #0000                ; Pointer: NIL
DEFB "E"+#80              ; Name
DEFB #02                  ; Typ: String
DEFB #03                  ; Inhalt: String-Laenge
DEFW #95E5                ;         String-Adresse

DEFW #0000                ; Pointer: NIL
DEFB "A","N","F"+#80      ; Name
DEFB #04                  ; Typ: Real
DEFB #00,#00,#30,#A3,#8f  ; Inhalt

DEFW #0009                ; Pointer --> Variable E
```

```
DEFB "E","N","D","E"+#80  ; Name
DEFB #04                  ; Typ: Real
DEFB #00,#00,#2C,#A3,#8F  ; Inhalt

DEFW #0000                ; Pointer: NIL
DEFB "D","P"+#80          ; Name
DEFB #04 +#40             ; Typ: Real, Bit 6 gesetzt
;                         ; => User-Funktion
DEFW #01CA                ; Zeiger auf Funktions-Definition

DEFW #0010                ; Pointer --> Variable ANF
DEFB "A"+#80              ; Name
DEFB #04                  ; Typ: Real
DEFB #00,#00,#00,#30,#8A  ; Inhalt
```

Wie an den beiden existierenden Hangel-Pointern in den Variablen ENDE und A nachgeprüft werden kann, zeigen die Pointer immer vor den ersten Buchstaben des Variablennamens oder, anders gesehen, auf das zweite, höherwertige Byte dessen Pointers. Das ist auch der Fall, wenn die Variablenadressen im BASIC-Programm eingetragen werden.

Dadurch ist übrigens sichergestellt, daß der Zeiger auf die erste Variable, die im Variablenbereich eingetragen ist, nicht &0000 sein kann (sondern &0001). Das ist wichtig, weil &0000 für NIL und für noch nicht eingetragene Pointer im Programmtext reserviert ist.

User Functions

Eine weitere interessante Sache ist, daß auch die User Functions im Variablenbereich eingetragen werden. Diese können genau wie die Variablen einen Typ zugeordnet bekommen. Im Fall der Doppel-Peek-Funktion hat der BASIC-Interpreter als Default den Typ Real angenommen. Um eine Funktion zu kennzeichnen, wird Bit 6 im Typen-Byte gesetzt.

Übrigens ist die Behandlung von User Functions nicht ganz fehlerfrei. Aber auch die Tatsache, daß die Functions einen Typ zugeordnet bekommen, kann zu Fehlern im Programmlauf führen, dann nämlich, wenn man globale Typendefinitionen erst nach der Funktionsdefinition ausführt:

```
10 DEF FNteste=12345
20 DEFINT s,t,u
30 PRINT FNteste
RUN [ENTER]
Unknown user function in 30
Ready
```

Der Grund ist einfach: Bei der Definition der Funktion wurde für den Namensanfang mit T noch standardmäßig Real angenommen. Danach wurde der

Default-Typ für S, T und U in Integer verändert. Beim Aufruf danach ohne explizite Typangabe nimmt BASIC den in Zeile 20 eingeführten Typ Integer für den Namensanfang T an und kann natürlich keine Integer-Funktion FNteste finden. Das Problem ist gelöst, wenn man entweder in Zeile 10 eine Integer-Funktion definiert oder in Zeile 30 explizit eine Real-Funktion aufruft:

```
10 DEF FNteste%=12345
20 DEFINT s,t,u
30 PRINT FNteste
RUN [ENTER]
 12345
Ready
```

```
10 DEF FNteste=12345
20 DEFINT s,t,u
30 PRINT FNteste!
RUN [ENTER]
 12345
Ready
```

```
10 DEFINT S,T,U
20 DEF FNteste=12345
30 PRINT FNteste
RUN [ENTER]
 12345
Ready
```

Dimensionierte Felder

Dimensionierte Variablen werden ähnlich wie die einfachen in einem eigenen Bereich abgespeichert, der hinter dem Bereich für normale Variablen folgt.

Daß für Felder ein eigener Bereich vorgesehen ist, hat wahrscheinlich seinen Grund darin, daß man mit dem Befehl ERASE einzelne Felder wieder löschen kann. Dadurch verschieben sich die Adressen der verbliebenen Felder innerhalb dieses Bereichs, und die im Programm eingetragenen Adressen müssen wieder korrigiert werden.

Weil die Felder aber nicht mit den normalen Variablen im selben Bereich abgespeichert sind, verändern sich die Adressen der normalen Variablen nicht. Hier müssen die bereits hergestellten Bindungen nicht korrigiert werden.

Auch innerhalb der Felder existieren verkettete Listen. Hier sind es aber nur drei, für jeden Datentyp eine.

Jedes Feld ist wie folgt aufgebaut:

```
; dimensionierte Variable
; ----------------------

DIMVAR: DEFW POINTER: Adresse des naechsten Feldes des gleichen Typs relativ zum
                      Start der Felder
        DEFM NAME des Feldes. Im letzten Buchstaben ist Bit 7 gesetzt.
        DEFB TYP: 1, 2 oder 4 für Integer, String oder Real.
        DEFW LAENGE des Feldes ab dem nächsten Byte.
        DEFB ANZAHL der Dimensionen
        DEFW erste Dimension (max. Index)
        ...
        DEFW letzte Dimension (max. Index)
        DEFS Platz für alle Daten dieses Feldes
```

Die Pointer, sowohl die der verketteten Listen als auch die im Programm eingetragenen, zeigen wieder auf das zweite Byte jedes Feldes, also auf das höherwertige Byte des Kettungs-Pointers. Auch diese Adressen sind wieder relativ zur Bereichsgrenze angegeben, in diesem Fall also zum Start des Bereichs der Felder.

Der Zugriff auf ein Datenelement in einem Feld ist nicht so leicht wie auf eine nicht dimensionierte Variable. Im Programm kann nur der Zeiger auf das gesamte Feld gespeichert werden. Auf Grund der Indizes, die ja bei jeder Bearbeitung des Befehls veränderbar sind, muß dann noch die Lage des gewünschten Feldelements berechnet werden. Die Felder, die BASIC einrichtet, sind Arrays aus Fixed Length Records. Der Datenzugriff in solchen Feldern ist im Kapitel über Datenspeicherung beschrieben.

Das folgende Beispiel zeigt die interne Speicherung einer BASIC-Zeile, in der auf Feldvariablen zugegriffen wird:

```
; BASIC-Zeile:   100 var%(0)=var(1)
; -----------   ------------------
ZEILE: DEFW 24              ; Zeilenlaenge
       DEFW 100             ; Zeilennummer
       DEFB #02             ; TOKEN: Integer-Variable mit %
       DEFW #0009           ;        Adresse
       DEFB "v","a","r"+#80 ;        Name
       DEFB "("             ;     \
       DEFB #0E             ;      > Indizes: (0)
       DEFB ")"             ;     /
       DEFB #EF             ; TOKEN: =
       DEFB #0D             ; TOKEN: Real-Variable ohne !
       DEFW #002A           ;        Adresse
       DEFB "v","a","r"+#80 ;        Name
       DEFB "("             ;     \
       DEFB #0F             ;      > Indizes: (1)
       DEFB ")"             ;     /
       DEFB 0               ; Endemarke
```

Wie man sieht, gibt es keine spezielle Kennzeichnung für dimensionierte Variablen. Der BASIC-Interpreter kann nur daran, daß eine Klammer-Auf "(" folgt, erkennen, daß es sich um eine Feldvariable handelt.

VERWALTUNG DER PROGRAMMSTRUKTUREN

Der BASIC-Interpreter benutzt einen eigenen Stapel, der nicht mit dem Returnstack der CPU identisch ist. Dieser Stapel ist 512 Bytes lang und wird bei der (rekursiven) Auswertung von arithmetischen Ausdrücken und für verschiedene Programm-Strukturen benutzt.

Der BASIC-Stack

Der Software-Stack liegt beim CPC 464 und 664/6128 an verschiedenen Stellen:

```
  +-----------------------+
  | BASIC-Software-Stack: |
  +-----------------------+

  CPC 464:

#AE8B STACK:  DEFS 512     ; reservierter Bereich für den Stack
#B08B STKPOI: DEFW #0000   ; Stapelzeiger

  CPC 664 / CPC 6128:

#AE6F: STACK:  DEFS 512    ; reservierter Bereich für den Stack
#B06F: STKPOI: DEFW #0000  ; Stapelzeiger
```

Der BASIC-Stack wächst von unten nach oben. Das folgende Programm schrieb das Grundgerüst für die danach folgende Darstellung des Stacks.

```
100 str=9
110 IF str=9 THEN OPENOUT"datei"
120 DEF FNdp(x)=PEEK(x)+256*PEEK(x+1)
130 stack =&ae6F
140 stkpoi=&B06F
150 FOR a=1 TO 1:GOSUB 160:NEXT                 <-- 1. FOR a!  2. GOSUB
160 ON BREAK GOSUB 170:WHILE 1:WEND             <-- 3. WHILE   4. ON BREAK GOSUB
170 FOR i%=UNT(stack) TO UNT(FNdp(stkpoi))      <-- 5. FOR i%
180   PRINT #str,"#";HEX$(i%);" #";HEX$(PEEK(i%),2)
200 NEXT
210 CLOSEOUT
```

Dabei ist nur die Darstellung der Einträge für die Programmstrukturen einfach möglich. Die Einträge bei der Auswertung von Ausdrücken sind von BASIC aus schlecht darstellbar, weil ja bei PEEK, PRINT usw. ebenfalls Ausdrücke ausgewertet werden und also auch hier der Stack benutzt wird. Es ist deshalb nicht so, daß sich während des Auslesens des Stack-Inhalts in der Zeile 180 auf dem Stack nichts mehr tut. Und auch die Bestimmung der Stapelspitze in Zeile 170 ergab den Wert, wie er bei der Auswertung des Ausdrucks UNT(FNdp(stkpoi)) gerade aktuell war.

```
; FOR-NEXT-Schleife (Real)

#AE6F DEFB #00                  ; Endemarkierung im Stapel (NIL)

#AE70 DEFW #02F5                ; Adresse der Schleifenvariable (FOR ...)
#AE72 DEFB #00,#00,#00,#00,#81 ; Endwert (TO ...)
#AE77 DEFB #00,#00,#00,#00,#81 ; Schrittweite (STEP ...)
```

```
#AE7C DEFB #01                  ; Vorzeichen der Schrittweite (STEP)
#AE7D DEFW #01F6                ; Zeiger hinter das FOR-Statement
#AE7F DEFW #01E8                ; Adresse der Zeile mit dem FOR-Statement
#AE81 DEFW #01FE                ; Zeiger hinter das NEXT-Statement
#AE83 DEFW #01FE                ; Zeiger hinter das NEXT-Token
#AE85 DEFB #16                  ; Laenge dieses Eintrags (gleichzeitig Kennung)

; GOSUB: Unterprogramm-Aufruf

#AE86 DEFB #00                  ; Kennung für GOSUB
#AE87 DEFW #01FC                ; Return-Adresse (Zeiger hinter GOSUB-Statement)
#AE89 DEFW #01E8                ; Adresse der Zeile mit dem GOSUB-Statement
#AE8B DEFB #06                  ; Laenge dieses Eintrags (und Kennung)

; WHILE-WEND-Schleife

#AE8C DEFW #0201                ; Adresse der Zeile mit dem WHILE-Statement
#AE8E DEFW #0210                ; Zeiger hinter das WEND-Statement
#AE90 DEFW #020D                ; Zeiger hinter das WHILE-Token
#AE92 07                        ; Laenge dieses Eintrags (und Kennung)

; Unterbrechung

#AE93 DEFB #02                  ; Kennung fuer ON BREAK GOSUB
#AE94 DEFW #AC17                ; Adresse des Parameterfeldes der Unterbrechung
#AE95 DEFW #0201                ; Adresse der Zeile, in der das Event auftrat
#AE98 DEFB #06                  ; Laenge des Eintrags (und Kennung)

; FOR-NEXT-Schleife (Integer)

#AE99 DEFW #02FE                ; Adresse der Schleifenvariablen (FOR ...)
#AE9B DEFW #AEB8                ; Endwert (TO ...)
#AE9D DEFW #0001                ; Schrittweite (STEP ...)
#AE9F DEFB #01                  ; Vorzeichen der Schrittweite
#AEA0 DEFW #0240                ; Zeiger hinter das FOR-Statement
#AEA2 DEFW #0213                ; Adresse der Zeile mit dem FOR-Statement
#AEA4 DEFW #02BB                ; Zeiger hinter das NEXT-Statement
#AEA6 DEFW #02BB                ; Zeiger hinter das NEXT-Token
#AEA8 DEFB #10                  ; Laenge dieses Eintrags (gleichzeitig Kennung)
```

Unterbrechungen

Unterbrechungen werden, wenn sie auftreten, vom BASIC-Interpreter fast wie ganz normale Unterprogramm-Aufrufe behandelt. Tatsächlich stellen sie das auch dar, nur, daß sie unvorhergesehen, an einer beliebigen Stelle des Programms auftreten können.

Das Betriebssystem unterscheidet zwischen "asynchronen" und "synchronen" Ereignissen. (Zum Interrupt-Mechanismus auf der Assembler- und BASIC-Ebene folgen noch eigene Kapitel.)

Der BASIC-Interpreter benutzt ausschließlich "synchrone" Interrupts. Das heißt, daß das Auftreten eines Interrupts noch nicht automatisch zum Aufrufen

des zugehörigen Unterprogramms führt. Das Vordergrund-Programm, also der BASIC-Interpreter, muß regelmäßig beim Betriebssystem nachfragen, ob ein Unterbrechungssignal aufgetreten ist.

Das macht BASIC immer zwischen zwei Statements. Tritt eine Unterbrechung auf, so kann sie so behandelt werden, als sei es ein Unterprogramm-Aufruf in dem gerade fertig bearbeiteten Befehl gewesen. Der Eintrag auf dem BASIC-Stack ist dem GOSUB-Eintrag sehr ähnlich, weist aber einen gravierenden Unterschied auf:

```
; Stack-Eintrag für ein Unterprogramm:

DEFB #00        ; Kennung: GOSUB
DEFW NXTBEF     ; Adresse des naechsten Befehls (Rueckkehradresse)
DEFW AKTZEI     ; Adresse der Zeile, in der der Aufruf erfolgte
DEFB #06        ; Kennung: Unterprogramm-Aufruf

; Stack-Eintrag für eine Unterbrechung:

DEFB #01        ; Kennung EVERY, AFTER oder ON SQ.  ON BREAK hat #02.
DEFW PARAMS     ; Adresse des Parameterfeldes der Unterbrechung
DEFW AKTZEI     ; Adresse der Zeile, in der der Aufruf erfolgte
DEFB #06        ; Kennung: Unterprogramm-Aufruf
```

Statt der Adresse des Befehls, zu dem mit RETURN zurückgekehrt werden muß, ist hier ein Zeiger auf einen Parameterblock eingetragen. Zu allen Interrupt-Quellen in BASIC gibt es einen zugehörigen Parameterblock:

	CPC 464	CPC 664/6128
ON BREAK GOSUB	&AC31	&AC17
ON SQ(1) GOSUB	&AC38	&AC1E
ON SQ(2) GOSUB	&AC44	&AC2A
ON SQ(4) GOSUB	&AC50	&AC36
AFTER/EVERY zeit,0 GOSUB	&AC5C	&AC42
AFTER/EVERY zeit,1 GOSUB	&AC6E	&AC54
AFTER/EVERY zeit,2 GOSUB	&AC80	&AC66
AFTER/EVERY zeit,3 GOSUB	&AC92	&AC78

Alle Parameterblöcke sind wie folgt aufgebaut:

```
PARAMS: DEFB Prioritaet des unterbrochenen Events (A-Register nach KL NEXT SYNC)
        DEFW Adresse des naechsten BASIC-Befehls (Rueckkehr-Adresse)
        DEFW Adresse des BASIC-Interrupt-Unterprogramms
```

Die Unterbrechungen werden systemkonform mit Event-Blocks u.ä. programmiert. Da es synchrone Events sind, wird vom Betriebssystem nur das Anstoßen (Kicken) eines Ereignisses (Events) vermerkt. Der BASIC-Interpreter schaut zwischen den Statements immer nach, ob ein Kick eingetroffen ist (mit HI KL POLL SYNC).

Wenn das der Fall ist, werden die Vektoren KL NEXT SYNC und KL DO SYNC aufgerufen. Das führt dazu, daß eine Interrupt-Behandlungsroutine des BASIC-Interpreters aufgerufen wird und diese einen Zeiger auf den zugehörigen Parameterblock übergeben bekommt.

Sie trägt nun in den BASIC-Stack einen Unterprogrammblock ein und verstellt den Programmzeiger im BASIC-Text auf das abzuarbeitende Unterprogramm. Danach wird die Bearbeitung des BASIC-Programms (ab der neu eingestellten Stelle) wieder ganz normal aufgenommen.

Schließt das BASIC-Interrupt-Programm mit "RETURN" ab, so findet die Behandlungsroutine für den Return-Befehl auf dem BASIC-Stack keinen normalen Unterprogrammblock, sondern einen Block für eine Unterbrechung; erkennbar an dem ersten Byte des Stack-Eintrags.

Die Rücksprungadresse muß nun also aus dem Parameterblock geholt und in den BASIC-Programmzeiger geladen werden. Außerdem wird aber auch noch der Vektor KL DONE SYNC aufgerufen und bei ON BREAK-Unterbrechungen der Break-Mechanismus des Betriebssystems wieder aktiviert.

Die verschiedenen Vektoren, die in diesem Zusammenhang aufgerufen werden, haben folgende Funktionen:

HI KL POLL SYNC	Teste, ob ein synchroner Interrupt angestoßen ist.
KL NEXT SYNC	Bestimme laufende Interrupt-Priorität, und erhöhe sie.
KL DO SYNC	Rufe Interrupt-Behandlungsroutine auf.
KL DONE SYNC	Restauriere alte Interrupt-Priorität.

Der Interrupt-Mechanismus des Betriebssystems ordnet den synchronen Events Prioritäten zu. Solange eine Unterbrechung aktiv ist, werden dem zwischen zwei Statements nachfragenden BASIC-Interpreter alle zwischendurch eingegangenen Events niedrigerer oder gleicher Priorität verheimlicht. Die verschiedenen Interrupt-Arten in BASIC haben dabei folgende Prioritäten:

ON BREAK GOSUB	Express, &40
AFTER/EVERY zeit,3	Normal, &10
AFTER/EVERY zeit,2	Normal, &08
ON SQ(x) GOSUB	Normal, &08
AFTER/EVERY zeit,1	Normal, &04
AFTER/EVERY zeit,0	Normal, &02
(kein Interrupt)	&00

Dabei werden die Prioritäten noch grundsätzlich in "normal" und "express" (dringend) unterteilt. Alle Express-Events sind dringender als alle "normalen".

Während also eine Unterbrechung des Timers Nummer 2 (Priorität normal, &08) läuft, sind alle Unterbrechungen durch Timer 0, 1 und die Unterbrechungen des Sound-Managers verboten, weil sie die gleiche oder niedrigere Priorität haben.

Trotzdem fragt BASIC zwischen den Statements auch weiterhin, ob mittlerweile weitere Unterbrechungssignale eingegangen sind. Das Betriebssystem informiert den BASIC-Interpreter aber nur noch über Unterbrechungen des Timers Nummer 3 (normal, &10) und über Break-Events (express).

Die BASIC-Befehle EI und DI rufen zwei Betriebssystem-Routinen auf, die die aktuelle Priorität vorübergehend so erhöhen, daß sie nur noch von Express-Events überboten werden kann. DI unterbindet also alle Timer- und Sound-Interrupts, nicht aber "Breaks".

Maschinencode auf dem CPC

Obwohl der Schneider CPC mit einem außerordentlich leistungsfähigen BASIC-Interpreter ausgestattet ist, ergibt sich doch früher oder später der Wunsch nach "Maschinencode".

Dabei muß man grundsätzlich zwei Fälle unterscheiden: Einmal kann man Assembler-Routinen als Unterprogramme einsetzen, die die Fähigkeiten des BASIC-Interpreters besonders bei zeitkritischen oder zeitintensiven Prozeduren erweitern. Zum anderen können aber auch Maschinencode-Programme als ein eigenständiges "Vordergrund-Programm" gestartet werden.

- Hintergrund-Programme = Systemerweiterungen
- Vordergrund-Programme = Hauptprogramme

Man kann dabei noch unterscheiden, wo der Maschinencode herkommen soll. Handelt es sich um ROM-Software, oder wird die Software von einem Massenspeicher geladen?

- ROM-Software
- Kassetten-/Disketten-Software

VORDERGRUNDPROGRAMME VOM MASSENSPEICHER

Zum Starten eines Hauptprogramms von Diskette oder Kassette sollte immer der Vektor &BD13 MC BOOT PROGRAMM benutzt werden. Dieser Vektor nimmt zunächst eine Teilinitialisierung des Rechners vor und ruft dann eine

Laderoutine auf, die man selbst bestimmen kann. Die Adresse der Laderoutine muß dem Vektor im HL-Register übergeben werden.

Die Laderoutine muß dann, wie es die Bezeichnung schon andeutet, das Assemblerprogramm in den Speicher des Rechners laden und kehrt danach mit "RET" zu MC BOOT PROGRAMM zurück.

Danach wird der Computer vollständig initialisiert und das Programm gestartet. Normalerweise wird man ein eigenständiges Maschinencode-Programm aber vom BASIC-Interpreter aus laden, weil dieser ja automatisch nach dem Einschalten des Computers gestartet wird. BASIC führt die beschriebenen Ladeaktionen aus, wenn man ein Maschinencode-Programm mit RUN"progname.ext" startet.

Die Aufrufbedingungen für das Maschinencode-Programm selbst sind dabei dieselben wie für jedes andere Vordergrundprogramm auch:

BC = &B0FF – letztes Byte des Memory Pools (total)
HL = &ABFF – Vorschlag für den statischen Variablenbereich des Vordergrundprogramms
DE = &0040 – erstes Byte des Memory Pools

Das RAM-Vordergrundprogramm muß sich dann zunächst seinen statischen Variablenbereich reservieren, danach evtl. Hintergrund-ROMs initialisieren (auch AMSDOS muß, wenn es benötigt wird, erst noch initialisiert werden!) und kann erst dann starten.

ROM-SOFTWARE

Alle ROMs mit zusätzlicher Software oder Vordergrundprogrammen müssen, wollen sie in den Genuß der existierenden Kernel-Routinen kommen, im obersten Adreßviertel der CPU eingeblendet werden.

Es können also zunächst einmal nur 16-kByte-Eproms, -PROMs oder -ROMs eingesetzt werden. Die belegten Adressen liegen im Bereich von &C000 bis &FFFF.

Dabei enthält der Schneider CPC einen Soft-/Hardware-Mechanismus, mit dem den einzelnen 16k-ROM-Blocks eine "ROM-Selekt-Adresse" im Bereich von &00 bis &FB (252) zugeordnet wird. (Die ROM-Selektion wurde bereits im Kapitel über die Speicherkonfiguration erläutert.)

Power on ROM

Die für das BASIC-ROM reservierte Adresse ist &00. Man kann aber auch ein anderes Vordergrund-ROM auf dieser ROM-Selekt-Adresse installieren. Das BASIC-ROM bleibt auf jeder noch nicht benutzten Adresse erreichbar, weil es über keine eigene Selekt-Decodierung verfügt und immer von anderen ROMs, die sich angesprochen fühlen, ausgeblendet werden muß. Der Kernel des Schneider CPC startet nach einem Kaltstart (Einschalten usw.) immer das Programm in dem ROM, das auf der Selekt-Adresse &00 eingeblendet wird.

Dadurch kann man seinen Computer beispielsweise zu einem Pascal-Rechner oder auch reinrassigen CP/M-Rechner machen. Im AMSDOS-Disketten-Controller ist ein Platz für eine Drahtbrücke vorgesehen, die, wenn man sie einsetzt, die Selekt-Adresse für das AMSDOS-ROM von &07 auf &00 ändert! Dadurch wird Amsdos anstelle des Basic-Interpreters gestartet. Die Initialisierungs-Routine von AMSDOS, das eigentlich ein "Hintergrund-ROM" ist, testet, ob Amsdos auf der Adresse &00 eingeblendet wurde und startet in diesem Fall CP/M!

ROM-Typen

Jeder ROM-Block hat einen sogenannten *Header*. Die ersten Bytes eines jeden ROMs müssen für Verwaltungsaufgaben geopfert werden:

```
#C000 DEFB Typ
#C001 DEFB Mark number
#C002 DEFB Version number
#C003 DEFB Modification level
#C004 DEFS External command table
 ....
```

Die drei Bytes auf den Adressen &C001 bis &C004 können vollkommen frei mit jedem beliebigen Wert beschrieben werden und sind nur für die eigene Statistik interessant.

Das erste Byte dagegen ist schon wichtiger. Der Kernel kennt drei verschiedene ROM-Typen und eine kleine Variation für das eingebaute BASIC-ROM:

```
     #C000 DEFB #00          ; Vordergrund-ROM
oder #C000 DEFB #00+#80      ; Eingebautes Vordergrund-ROM
oder #C000 DEFB #02          ; Erweiterungs-ROM
oder #C000 DEFB #01          ; Hintergrund-ROM
```

– &80

Im Typ-Byte des BASIC-ROMs ist Bit 7 gesetzt. Das "On-Board-ROM" hat keine eigene Decodierungsschaltung für seine ROM-Selekt-Adresse. Jeder Lesezugriff auf ein ROM bezieht sich automatisch auf das BASIC-ROM, wenn es nicht von einem anderen ROM, das sich angesprochen fühlt, explizit mit der Leitung ROMDIS ausgeblendet wird.

Wenn der Kernel alle ROM-Adressen nach Hintergrund-ROMs absucht, erkennt er am Typ-Byte &80 auf der Adresse #C000, daß auf dieser ROM-Select-Adresse kein anderes ROM installiert ist, weil hier das BASIC-ROM "durchschimmert".

– &00

ROMs, die Vordergrundprogramme enthalten, werden mit einem Byte &00 auf der Adresse &C000 gekennzeichnet.

– &02

Sehr umfangreiche Vordergrundprogramme können bis zu vier 16k-Blocks umfassen. Diese müssen dann aufeinanderfolgende Selekt-Adressen haben. Nur das erste ROM darf in seinem Header als Vordergrund-ROM gekennzeichnet sein. Die bis zu drei "nachfolgenden" ROMs müssen als Erweiterungs-ROM deklariert werden.

Vor allem mit Restart_2 ist es dabei möglich, von jedem ROM eines Hauptprogramms aus Unterprogramme in "benachbarten" Erweiterungs-ROMs aufzurufen.

– &01

Alle ROMs, die keine eigenständige Software enthalten, sondern nur zusätzliche Routinen, die von einem Vordergrundprogramm benutzt werden können, müssen als Hintergrund-ROM markiert werden.

Alle Hintergrund- ROMs sollten auf Selekt-Adressen von &01 bis &07 (CPC 464) bzw. &00 bis &0F (CPC 664, 6128) installiert werden, weil der Vektor KL ROM WALK nur diese ROM-Adressen abfragt und auch nur für diese ROMs Zeiger auf deren reservierten RAM-Bereich speichern kann. Das AMSDOS-ROM hat die Selekt-Adresse &07.

HINTERGRUND-ROUTINEN VON MASSENSPEICHERN

Hintergrund-Software, die erst nachträglich von Kassette oder Diskette geladen wird, muß vom Vordergrundprogramm ihren Platz zugewiesen bekommen.

CPC-typisch ist dabei die Methode, daß das Vordergrundprogramm von seinem Memory Pool von oben her etwas abtritt. Beim eingebauten BASIC- Interpreter müßte man also HIMEM herabsetzen und danach die Datei mit dem Assemblerprogramm dorthin laden.

Problematisch ist für das geladene Maschinencode-Programm vor allem, daß nicht vorhersehbar ist, an welche Adresse es genau zu liegen kommt. Das RAM wird ja dynamisch von oben und von unten her den verschiedenen Erweiterungen zugeteilt, wobei die Grenzen des noch verfügbaren Memory Pools langsam aufeinander zu wachsen.

Da Maschinencode-Programme, bis auf ganz wenige Ausnahmen, nicht ortsunabhängig geschrieben werden können, müssen sie der realen Lage angepaßt, also "relocalisiert" werden. Am Ende dieses Kapitels wird deshalb ein einfacher "Relocater" für Z80-Assemblerprogramme beschrieben. Dieses Zusatzprogramm muß aufgerufen werden, bevor die erste Routine in der Erweiterung benutzt werden kann. Dabei paßt es alle absoluten Adressen an die aktuelle Lage des Programms an. Im Anschluß an die Verschiebung können noch weitere Initialisierungen vorgenommen werden, zum Beispiel die Einbindung als RSX.

RESIDENT SYSTEM EXTENSIOS

Der Kernel des Schneider CPC unterstützt ein System externer Kommandos (RSX), bei der alle verfügbaren Zusatzroutinen über einen Namen ansprechbar sind. Von BASIC aus sind diese Kommandos mit Hilfe des "Extended Colon" genannten Zeichens zugänglich, beispielsweise:

```
|BASIC [ENTER]
```

In dieses System sind alle Vordergrund-ROMs und alle initialisierten Hintergrund-ROMs eingebunden und ebenfalls alle nachgeladenen Maschinencode-Programme, die sich über den Vektor &BCD1 KL LOG EXT dem Betriebssystem vorgestellt haben.

Dafür wird für jedes Programmpaket eine "External Command Table", also eine Tabelle der externen Kommandos benötigt. Die einzelnen Tabellen wer-

den über eine verkettete Liste "zusammengehalten". Zusätzlich zu jeder Kommandotabelle werden vier Bytes für diesen Kettungsmechanismus benötigt.

Bei Vorder- und Hintergrund-ROMs ist die External Command Table Bestandteil des ROM-Headers. Wird ein Hintergrund-ROM initialisiert, so darf es sich vom Memory Pool etwas Speicherplatz reservieren (siehe Kapitel über die Speicheraufteilung der CPC-Rechner). Dabei zieht der Kernel die vier Bytes für die verkettete Liste einfach noch zusätzlich vom Memory Pool ab.

Mit dem Vektor &BCD4 KL FIND COMAND kann zu einem Kommandonamen dessen Adresse und ROM-Selekt-Byte erfragt werden. Dabei werden die Kommandotabellen von hinten her durchsucht. Die RSX-Erweiterungen, die zuletzt eingeführt wurden, werden als erste überprüft. Sollten also zwei Routinen mit dem gleichen Namen existieren, so wird die ältere von der später angefügten Routine "verdeckt".

Beim Suchen eines Kommandos werden zuerst alle RSX-Erweiterungen im RAM abgesucht, dann alle Hintergrund-ROMs und, wenn der Befehl immer noch nicht gefunden wurde, auch noch alle Vordergrund-ROMs ab der ROM-Selekt-Adresse &08 (beim CPC 464) bzw. &10 (bei den CPCs 664 und 6128), bis der Kernel einen unbenutzten ROM-Steckplatz findet.

Der Vektor KL FIND COMMAND ruft dabei die gefundene Routine nicht selbst auf, sondern gibt nur die Routinenadresse in HL und die ROM-Konfiguration in C zurück. Ausgenommen ist nur der Fall, daß ein Vordergrund-Kommando, wie etwa "|BASIC" aufgerufen werden soll. In diesem Fall wird der Computer initialisiert und das Programm direkt gestartet.

Sonst muß man die entsprechende Routine noch via Restart 3 (LOW FAR CALL) oder über &001B LOW FAR PCHL aufrufen. Dabei wird allen Routinen in Hintergrund-ROMs die Startadresse des von ihnen über dem Memory Pool reservierten Speicherbereichs im IY-Register übergeben.

External Command Table

Nach diesen Einzelheiten bleibt nur noch das Layout der External Command Tables zu klären:

```
; Aufbau der Namenstabellen fuer RSX-Kommandos:
; ---------------------------------------------
RSXTAB: DEFW NAMTAB    ; Zeiger auf Namenstabelle
        JP   ROUT1     ; \
        JP   ROUT2     ;  \  Sprungleiste zu den Routinen;
        JP   ROUT3     ;   > koennen natuerlich auch mit
```

```
        ...             ;  /  Restarts gebildet werden.
        JP    ROUTn     ; /
;
NAMTAB: DEFB "R","O","U","T","1"+#80
        DEFB "R","O","U","T","2"+#80
        DEFB "R","O","U","T","3"+#80
        ..
        DEFB "R","O","U","T","n"+#80
        DEFB 0
```

Dabei müssen die Sprungvektoren in der Reihenfolge der Namen in der Namenstabelle aufeinanderfolgen. In der Namenstabellen werden die Namen der einzelnen RSX-Befehle in ihrer ASCII-Codierung abgelegt. Im letzten Buchstaben eines Namens muß jeweils das 7. Bit gesetzt sein. Abgeschlossen wird die Tabelle durch ein Null-Byte.

Bei Hintergrund-ROMs wird der erste Eintrag benutzt, um das ROM zu initialisieren. Hier sollte der Name möglichst so gestaltet werden, daß man ihn nicht von BASIC aus unbeabsichtigt aufrufen kann. Das kann man erreichen, indem man beispielsweise Kleinbuchstaben, Spaces oder Sonderzeichen in den Namen einbaut (bei der Namensgebung ist vom Kernel her fast alles erlaubt).

Auch kann man von einem Vordergrundprogramm aus gut testen, ob ein spezielles Hintergrund-ROM bereits initialisiert ist, indem man mit KL FIND COMMAND versucht, die Adresse des Initialisierungsvektors zu ermitteln. Man muß dazu nur den Namen des ersten Vektors kennen. Um in einer Mcode-Routine zu erfahren, ob AMSDOS initialisiert ist, könnte man mit KL FIND COMMAND die Adresse des Kommandos CPM ROM suchen lassen.

Vordergrund-ROMs sollten normalerweise nur einen Eintrag haben: den, mit dem sie gestartet werden. Nur wenn ein ROM mehrere voneinander unabhängige Vordergrundprogramme enthält, können in der Kommandotabelle auch mehrere Einträge auftauchen.

Als praktische Beispiele folgen nun Assemblerprogramme für eine RAM-RSX und ein ROM-Header:

```
; RAM-RSX-Erweiterung z.B. fuer einen Drucker-Spooler:
; ---------------------------------------------------
INIT:   LD HL,SPACE    ; Adresse der 4 freien Bytes fuer die
;                      ; Command Chain
        LD BC,RSXTAB   ; Adresse der External Command Table
        CALL #BCD1     ; und KL LOG EXT aufrufen
        RET

SPACE:  DEFS 4

RSXTAB: DEFW NAMTAB    ; Adresse der Namenstabelle
```

```
        JP   ON        ; Vektoren zu den einzelnen
        JP   OFF       ; RSX-Routinen
        JP   FLUSH
        JP   ASKFRE

NAMTAB: DEFB "S","P","O","O","L",".","O","N"+#80      ; SPOOL.ON
        DEFB "S","P","O","O","L",".","O","F","F"+#80  ; SPOOL.OFF
        DEFB "F","L","U","S","H",".","B","U","F"+#80  ; FLUSH.BUF
        DEFB "A","S","K",".","F","R","E","E"+#80      ; ASK.FREE
        DEFB 0


; ROM-Header z. B. fuer eine Grafikerweiterung
; ------------------------------------------
#C000:  DEFB 1         ; Typ = Hintergrund-ROM
        DEFS 3         ; Mark, Version, Modification

RSXTAB: DEFW NAMTAB    ; Zeiger auf die Namenstabelle.
#C006:  JP   INIT      ; Initialisierungsroutine (immer auf
;                      ; dieser Adresse!)
        JP   BOX       ; Vektoren zu den einzelnen
        JP   CIRCLE    ; RSX-Routinen
        JP   ELLIPSE
        JP   FILL
        JP   TRIANG

NAMTAB: DEFB "G","r","a","f","i","k"+#80         ; Name des Hinter-
;                                                ; grund-ROMs
        DEFB "B","O","X"+#80                     ; BOX
        DEFB "C","I","R","C","L","E"+#80         ; CIRCLE
        DEFB "E","L","L","I","P","S","E"+#80     ; ELLIPSE
        DEFB "F","I","L","L"+#80                 ; FILL
        DEFB "T","R","I","A","N","G","L","E"+#80 ; TRIANGLE
        DEFB 0
```

Der Aufruf einer RSX-Erweiterung gestaltet sich in Assembler nicht ganz so einfach wie in BASIC. Hier muß man erst mit KL FIND COMMAND die Rout- ine suchen und dann auch noch selbst aufrufen. Das folgende Programm ist ein Beispiel für einen RSX-Aufruf in Maschinensprache. Soll ein bestimmtes ex- ternes Kommando sehr oft aufgerufen werden, so empfiehlt es sich jedoch, die Adresse nur einmal zu bestimmen.

```
; Aufruf einer RSX in Maschinensprache
; ------------------------------------
        ...
        LD   HL,NAME          ; Zeiger auf den Namen nach HL laden
        CALL #BCD4            ; KL FIND COMMAND
        JR   NC,FEHLER        ; NC -> nicht gefunden !!

        LD   (FARADR),HL      ; FAR ADDRESS fuer Restart 3 basteln:
        LD A,C                ; zuerst Routinenadresse in HL
        LD (FARADR+2),A       ; und dann ROM-Konfigurationsbyte in C
```

```
        RST #18              ; mit Restart 3: FAR CALL
        DEFW FARADR          ; das externe Kommando aufrufen
        ...                  ; und weiter im Programm ...
        ...

FARADR: DEFS 3               ; Platz fuer die FAR ADDRESS zum Restart 3
        ...
NAME:   DEFB "D","I","R"+#80 ; Name der zu rufenden RSX; letzter
;                            ; Buchstabe
                             ; mit gesetztem 7. Bit
```

BASIC und Maschinencode

Sollen Assemblerroutinen und BASIC zusammen im Speicher des Computers existieren, so muß der BASIC-Interpreter das Vordergrundprogramm sein. Die Assemblerroutinen können nur als Hintergrundprogramme zusätzliche Funktionen bereitstellen.

Alle Hintergrund-ROMs werden von BASIC automatisch initialisiert. Der BASIC-Interpreter ruft dafür, sobald ihm vom Kernel die Kontrolle übergeben wurde, den Vektor &BCCB KL ROM WALK auf. Auch AMSDOS wird auf diese Weise initialisiert, wenn es vorhanden ist.

Darüber hinaus können Zusatzprogramme natürlich auch noch von Kassette oder Diskette nachgeladen werden. Dabei muß man den Hintergrund-Routinen selbst genügend Platz zuweisen.

Maschinencode im String

Eher selten wird dabei die Möglichkeit benutzt, Mcode-Routinen in ein String zu laden. Der Grund dafür ist, daß die Lage von Strings nicht nur nicht vorhersehbar ist, sondern sich nach einer Garbage Collection auch ändern kann!

Routinen in Strings müssen deshalb ortsunabhängig sein, was bei Assemblerprogrammen nur in den seltensten Fällen zu erreichen ist.

Außerdem stehen noch einige andere Einschränkungen und Probleme im Weg: So ist beispielsweise die maximale Länge solcher Programme durch die maximale String-Länge auf 255 Bytes festgelegt.

Aber das ist eigentlich nicht so schlimm, da ortsunabhängige Z80-Programme in aller Regel sowieso nur bei ganz einfachen Routinen zu realisieren sind.

Auch bereitet es zunächst einmal Schwierigkeiten, den Maschinencode überhaupt in den String hineinzubekommen. Die folgende Methode ist leider in den meisten Fällen ungeeignet:

```
OPENIN "mcode.dat"
LINE INPUT#9, mcode$
CLOSEIN
```

Das liegt daran, daß die ASCII-Zeichen mit dem Code 13 und 26 als Steuerzeichen aufgefaßt und Null ganz ignoriert und nicht eingelesen wird. Die Chance, daß eine Assemblerroutine rein zufällig diese Bytes in ihrem Programmcode enthält, ist ziemlich hoch.

Man muß deshalb einen Kunstgriff anwenden, um den Maschinencode doch noch in den String zu bekommen.

Eine Möglichkeit ist, ihn aus DATA-Zeilen auszulesen und in einen String zu POKEn. Der Platzverbrauch für DATA-Zeilen ist aber immer unverhältnismäßig hoch.

Man kann die Programmdatei aber auch trotzdem von Diskette laden:

```
10 adr=HIMEM-255:MEMORY adr-1
20 LOAD "mcode.bin",adr
30 mcode$="": POKE @mcode$ ,prog.laenge          ← String-Laenge
              POKE @mcode$+1,adr-256*INT(adr/256)  ← LSB der String-
                                                     Adresse
              POKE @mcode$+2, INT(adr/256)         ← MSB der String-
                                                     Adresse
40 mcode$=mcode$+""
50 MEMORY adr+255
```

In Zeile 10 wird zunächst über HIMEM genügend Platz freigemacht, um die Programmdatei aufzunehmen. Diese wird dann in Zeile 20 auch dorthin geladen.

In Zeile 30 wird der String dann eingerichtet und der String-Descriptor, dessen Adresse man mit der Funktion @ erhält, so umgePOKEt, daß er auf den Maschinencode zeigt. Durch die Berechnung eines "neuen" Mcode-Strings in Zeile 40 wird der String im Memory Pool neu angelegt. Das ist wichtig, damit der String die nächste Garbage Collection überlebt.

In Zeile 50 wird zum Schluß der Puffer über HIMEM wieder geschlossen.

Man hat aber nicht nur Probleme, den Mcode in den String hineinzubekommen, auch sein Aufruf erweist sich als problematisch. Wenn nicht mit viel

Aufwand eine Garbage Collection sicher auszuschließen ist, muß vor jedem Aufruf der String-Routine die Adresse neu berechnet werden. Das kann mit der folgenden Funktion aber noch recht komfortabel erreicht werden:

```
60  DEF FNmcode=PEEK(@mcode$+1)+256*PEEK(@mcode$+2)
    ....
    ....
500 CALL FNmcode
```

Die Funktion FNmcode PEEKt sich die aktuelle Adresse des Strings immer aus dem String-Descriptor heraus. Sind mehrere String-Routinen vorhanden, so kann für jede eine eigene Funktion definiert werden.

Die String-Methode hat aber auch Vorteile: So kann man Überschneidungen zwischen verschiedenen Routinen sicher ausschließen, und Routinen, die nicht mehr benötigt werden, können jederzeit aus dem Speicher gelöscht werden:

```
999 mcode$=""
```

MASCHINENCODE ÜBER HIMEM

CPC-typisch ist aber die Methode, für den Maschinencode über HIMEM etwas Speicherplatz zu opfern. Zwar ist die Ladeadresse für das Assemblerprogramm zunächst auch nicht vorhersehbar. Sobald es geladen ist, wird es aber nicht mehr verschoben. Es ist deshalb nicht nötig, den Maschinencode vollkommen ortsunabhängig zu gestalten. Es genügt, wenn man das Programmpaket einmal initialisiert und dabei alle absoluten Adressen der tatsächlichen Lage des Programms anpaßt, es also "relocalisiert".

Ausgesprochen schlecht beraten ist man, wenn man für eine universell verwendbare Erweiterung eine feste Adresse im RAM annimmt. Beispielsweise könnte man annehmen, daß in BASIC HIMEM immer über 40000 liegen wird, und dann das Programm ab der Adresse 40000 assemblieren.

Das geht nur so lange gut, wie man auf diese Idee kein zweites Mal verfällt und beide Erweiterungen gleichzeitig zur Verfügung haben will.

Man sollte auch immer bedenken, daß zu jeder Erweiterung möglicherweise eine Treibersoftware in einem Hintergrund-ROM gehört. Erstes Beispiel ist hier AMSDOS mit einem Speicherplatzverbrauch von nicht weniger als 1280 Bytes! Da könnte irgendwann einmal HIMEM für die Mcode-Erweiterung mit fester Adresse zu niedrig werden.

CALL

Im Schneider-BASIC ist es zwar möglich, Maschinencode durch Angabe der Einsprungsadresse aufzurufen. Da diese zunächst aber unbekannt ist, sollten dafür Variablen vorsehen werden:

```
10  HIMEM=HIMEM-prog.länge           ← Speicher reservieren
20  adr=HIMEM+1:LOAD"grafik.bin",adr ← Programmdatei laden
30  CALL adr                         ← Initialisieren, verschieben
40  box      = adr+&123              ┐
50  circle   = adr+&234              } Variablen für die verschiedenen
60  triangle = adr+&345              ┘ Einsprungadressen definieren
     ....
     ....
500 CALL box, 100,200,300,400        ← Aufruf
```

RSX

Darüber hinaus können die Mcode-Routinen bei ihrer Initialisierung als RSX eingebunden werden. Dann stehen sie in BASIC direkt mit einem Namen zur Verfügung:

```
10   HIMEM=HIMEM-prog.laenge
20   adr=HIMEM+1:LOAD"grafik.bin",adr
30  CALL adr              ← Verschieben, Initialisieren,
                                        Einbinden als RSX
     ....
500 |BOX, 100,200,300,400 ←Aufruf eines RSX-Kommandos
```

RSX-Kommandos werden in BASIC mit einem "Extended Colon" (|) markiert. Dieses Zeichen ist jedoch kein Bestandteil des RSX-Namens!

Dabei sind die beiden Methoden, eine Mcode-Routine aufzurufen, fast vollkommen gleichwertig. Ein Unterschied ist jedoch, daß bei RSX-Paketen das BASIC-Programm die Lage der Einsprungadressen nicht zu wissen braucht. Verschieben sich die Einsprungadressen, weil man im Mcode Änderungen vorgenommen hat, so müssen bei RSX-Erweiterungen in den BASIC-Programmen keine Aufrufadressen geändert werden, wohl aber bei Aufrufen mit CALL.

Außerdem können RSX-Erweiterungen auch in ROMs enthalten sein (und werden dann von BASIC automatisch initialisiert). Aufrufe mit CALL sind nur im RAM möglich.

PARAMETER

Wird eine Assemblerroutine aufgerufen, so können an sie bis zu 32 Parameter übergeben werden. Dabei ist es egal, ob der Aufruf mit RSX oder mit CALL erfolgte:

```
CALL box, 100,200,300,400
    |BOX, 100,200,300,400
```

Als Parameter kommen allerdings nur Integer-Werte in Frage, und die Parameterübergabe funktioniert auch nur in eine Richtung: von BASIC an das Assemblerprogramm. Will man mehr erreichen, so muß man tricksen.

Jeder Parameter kann im Bereich von –32768 bis +65535 liegen (im Firmware-Manual wird als Obergrenze +32767 angegeben, was aber nicht stimmt). Diese werden als 16-Bit-Words an die Mcode-Routine übergeben.

Integer-Variablen sind 16-Bit-Words mit Vorzeichen und können Zahlenwerte von –32768 bis +32767 darstellen. Es ist aber auch möglich, Words ohne Vorzeichen im Bereich von 0 bis 65535 zu übergeben. Dabei ist auf die Doppeldeutigkeit der negativen Zahlen zu achten:

```
Integer (mit VZ):    0 - 32767 /-32768 - -1
ohne Vorzeichen:     0 - 32767 / 32768 - 65535
```

Die Übergabe an die Maschinencode-Routine wird vom Schneider-BASIC wie folgt realisiert. Diese Methode ist zwar nicht verbindlich und wegen ihrer Einschränkungen auch nicht besonders elegant. Trotzdem sollte sie, um die Kompatibilität zu wahren, auch von anderen Vordergrundprogrammen folgendermaßen gehandhabt werden:

Parameterübergabe

Im A-Register wird der Routine die Anzahl der übergebenen Parameter mitgeteilt. Diese Parameter befinden sich auf einem Stapel (und zwar auf dem Hardware-Stack, was für den Anwender aber uninteressant ist). Der erste Parameter liegt dabei an der obersten Adresse, und das IX-Register zeigt auf das LSB (unterstes Byte) des letzten Parameters (unterster Stapeleintrag):

```
Register        Speicher
-------------   -------------
A = n           MSB PARAM1
                LSB PARAM1
                MSB PARAM2
                LSB PARAM2
```

```
                        ...
                        ...
                        ...
                        MSB PARAMn
IX ---------->          LSB PARAMn
```

Parameterübernahme

Die Mcode-Routine kann nun über das IX-Register auf die übergebenen Parameter zugreifen:

```
ROUT1:  CP   2          ; Test, ob Anzahl Parameter stimmt (hier 2)
        RET  NZ         ; nein: Fehler
        LD   L,(IX+0)
        LD   H,(IX+1)   ; HL = letzter (= zweiter) Parameter
        LD   E,(IX+2)
        LD   D,(IX+3)   ; DE = vorletzter (= erster) Parameter
        ...
```

Daß das IX-Register immer auf den letzten übergebenen Parameter zeigt, ist für all diejenigen Routinen ungünstig, die eine veränderliche Anzahl an Argumenten übernehmen können. Dabei sind nämlich in aller Regel die hinteren Parameter optional. Der Zugriff auf die tatsächlich im Aufruf angegebenen Parameter ist dann erschwert, weil hier zum IX-Register ein anderer Index addiert werden müßte. Das Problem kann man wie folgt angehen:

```
; Beispiel: Routine nimmt 2 oder 3 Parameter:

ROUT2:  CP   2          ; Test, ob mind. 2 Argumente
        RET  C          ; Nein: Fehler

        LD   BC,#0001   ; Default-Annahme fuer dritten Parameter machen
        CP   3          ; Test, ob dritter Parameter angegeben ist
        JR   C,ROUT2A   ; nein, nur zwei -> Default uebernehmen

        RET  NZ         ; Fehler, falls nicht 3 (also mehr als 3)
;                       ; Parameter

        LD   C,(IX+0)
        LD   B,(IX+1)   ; letzten (dritten) Parameter uebernehmen

        INC  IX         ; und jetzt so tun, als seien nur zwei Parameter
        INC  IX         ; uebergeben worden

ROUT2A: LD   E,(IX+0)   ; uebernehme letzten (zweiten) Parameter
        LD   D,(IX+1)
        LD   L,(IX+2)   ; uebernehme vorletzten (ersten) Parameter
        LD   H,(IX+3)
        ...
```

Übergabe von Fließkommazahlen

Obwohl man an Mcode-Routinen eigentlich nur Integer-Zahlen übergeben kann, ist es mit einem kleinen Trick trotzdem möglich, auch Real-Zahlen oder Strings zu übergeben. Man speichert den Wert einfach in einer Variablen und übergibt dann deren Adresse an das aufgerufene Programm:

```
100 a!=123.45
110 b!=234.56
120 |RSX,@a!,@b!


ROUT3:  CP   2
        RET  NZ          ; Parameterzahl kontrollieren

        LD   E,(IX+0)
        LD   D,(IX+1)    ; DE = @b
        LD   L,(IX+2)
        LD   H,(IX+3)    ; HL = @a
        ...
```

Da die Rechenroutinen der Fließkomma-Arithmetik sowieso immer die Zeiger auf die zu bearbeitenden Zahlen benötigen, ist das geradezu ideal. Man muß nur beachten, daß die Variablen möglicherweise (und bei kurzen BASIC-Programmen ziemlich sicher) im untersten Speicherviertel liegen. Die Rechenroutinen blenden aber das untere RAM aus, weil sie selbst im Betriebssystem-ROM liegen. Die Routinen der Fließkomma-Arithmetik funktionieren nicht im unteren RAM-Viertel. Hier muß man die Zahlen zunächst immer in Zwischenspeicher über &4000 kopieren, bevor man mit ihnen rechnen kann.

Übergabe von Strings

Bei Strings funktioniert es genauso, nur daß man jetzt die Adresse des Descriptors einer String-Variablen übergeben muß. Dabei wurde beim 664/6128-BASIC eine Vereinfachung vorgenommen. Beim CPC 464 muß man immer den Klammeraffen @ vor den Variablennamen stellen, um die Übergabe der String-Descriptor-Adresse zu erreichen. Beim CPC 664 und 6128 ist das nicht mehr nötig:

```
100 a$="TESTTESTTEST"
110 |RSX,@a$            ← CPC 464
110 |RSX, a$            ← CPC 664/6128


ROUT4:  CP   1
        RET  NZ          ; Parameterzahl testen
```

```
        LD   L,(IX+0)
        LD   H,(IX+1)    ; HL = @a$ = Adresse des Descriptors

        LD   A,(HL)      ; A = Laenge des Strings
        INC  HL
        LD   E,(HL)
        INC  HL
        LD   D,(HL)      ; DE = Adresse des Strings
        ...
```

Übernahme von Ergebnissen

Mit Hilfe des Variablen-Pointers @ ist es auch möglich, wieder Ergebnisse von einer Assemblerroutine zurückzubekommen. Man übergibt, eventuell zusätzlich zu den Parametern für die Routine, die Adresse einer Rücknahmevariablen. Diese kann von jedem beliebigen Typ sein. Der Typ muß nur mit dem Unterprogramm eindeutig vereinbart werden, weil es in der Mcode-Routine keine Möglichkeit gibt, zu überprüfen, was man an Parametern übergeben bekommen hat:

```
|RSX ,100 ,@i% ,@a! ,@s$
```

einen Zahlenwert oder
die Adresse einer Integer-Variablen oder
die Adresse einer Real-Variablen oder
die Adresse einer String-Variablen

Das ist besonders kritisch, wenn das Maschinencode-Programm die adressierte Variable ändern will. Stimmt der Typ nicht, oder ist es noch nicht einmal die Adresse einer Variable, kann die Speicherorganisation des BASIC-Interpreters recht schnell und recht gründlich durcheinandergeraten.

Übernahme eines Integer-Ergebnisses:

```
100 i%=0
110 |RSX,100,200,@i%
120 PRINT i%
```

```
RSX5:   CP   3
        RET  NZ          ; Parameterzahl ueberpruefen.

        LD   L,(IX+2)
        LD   H,(IX+3)    ; zweitletzter = zweiter Parameter
        LD   E,(IX+4)
        LD   D,(IX+5)    ; drittletzter = erster Parameter
```

```
        ADD   HL,DE       ; Funktion dieser RSX: addiere [1] + [2] -> [3]
        EX    DE,HL       ; Ergebnis nach DE

        LD    L,(IX+0)
        LD    H,(IX+1)    ; HL = Adresse von i%

        LD    (HL),E
        INC   HL
        LD    (HL),D      ; Ergebnis in i% abspeichern

        RET               ; fertig
```

Bei Fließkommazahlen und Strings funktioniert es entsprechend, nur daß man hier 5 bzw. 3 Bytes ändern muß. Ändert man bei Strings aber nicht den Descriptor, sondern den String-Inhalt, so muß man beim Aufruf von solchen Routinen darauf achten, daß der Descriptor nicht mehr in das BASIC-Programm selbst zeigt. Das ist bei allen "direkten" Definitionen der Fall, bei denen der BASIC-Interpreter nicht zu rechnen braucht:

```
100 LET a$="abcde"          ' ← Descriptor zeigt in den Programmtext!!

100 LET a$="abcde"+""       ' ← BASIC muss rechnen → String liegt im
                                String-Bereich
```

HILFREICHE FEHLERMELDUNGEN

Nach einiger Zeit kann es schon vorkommen, daß man nicht mehr weiß, welche Parameter in welcher Reihenfolge an eine Mcode-Routine übergeben werden müssen. Es wäre also eine Meldung wünschenswert, die die genaue Syntax des Aufrufes beschreibt, etwa so:

```
'USE: |FILL ,x ,y [,farbe]
```

Das ist ganz einfach zu erreichen, und es gibt eigentlich keinen Grund (außer einem geringen Mehr-Verbrauch an Speicherplatz), wieso man darauf verzichten sollte. Das folgende Programm zeigt, wie man so etwas realisieren kann:

```
; Assembler-Routine mit Fehlermeldungen (Syntax-Hinweisen)
; z.B.: |FILL-Routine mit 2 Parametern:

FILLXY: LD    HL,TEXT6      ; HL mit Zeiger auf Fehlermeldung laden
        CP    2
        JP    NZ,FEHLER     ; falls keine 2 Parameter
        ...
        ...
```

```
; Routine druckt Meldung ab (HL) aus:
FEHLER: LD   A,(HL)
        AND  A              ; Textende?
        RET  Z
        CALL #BB5A          ; TXT OUTPUT: Drucke Zeichen oder befolge
;                           ; Controlcode
        INC  HL
        JR   FEHLER

TEXT6:  DEFB 13             ; Cursor zum der Zeilenanfang
        DEFB 18             ; Zeile loeschen
        DEFB 10             ; Cursor in die naechste Zeile
        DEFM "USE: |FILL ,x-Koordinate ,y-Koordinate"
        DEFB 18             ; Rest der Zeile loeschen
        DEFB 13             ; zurueck zum Zeilenanfang
        DEFB 10             ; und eine Zeile tiefer
        DEFB 18             ; Zeile loeschen
        DEFB 10             ; und noch eine Zeile tiefer
        DEFB 0              ; Ende des Textes


|FILL [ENTER]

USE: |FILL ,x-Koordinate ,y-Koordinate

Ready
```

Beim Aufruf der Routine FILLXY (via |FILL) wird zunächst ein Zeiger auf den Fehlertext geladen.

Danach wird die Anzahl der Parameter überprüft. Falls in diesem Beispiel nicht genau zwei Parameter übergeben wurden, wird zur Ausdruckroutine "FEHLER" verzweigt, die den durch HL angezeigten Text ausgibt und danach zurückkehrt. In diesem Beispiel wurde die Fehlermeldung mit vielen Controlcodes versehen, damit der Text auch gut sichtbar wird.

RSX-LOADER

Alle Programmpakete, die RSX-Befehle beinhalten (die meisten anderen aber auch) müssen initialisiert werden, bevor sie benutzt werden können. Danach läßt sich das gesamte Programmpaket in die zur Verfügung gestellten Routinen und in die nun nutzlose Initialisierungsroutine trennen.

Die Initialisierungsroutine wird normalerweise nicht mehr benötigt (und darf in aller Regel kein zweites Mal aufgerufen werden). Der Platz, den sie beansprucht, kann also vom Vordergrundprogramm (BASIC) wieder für andere Zwecke benutzt werden.

Es ergibt sich somit folgender Ablauf, bis ein Utility eingebunden ist:

1. Speicherplatz reservieren (über HIMEM)
2. Maschinencode laden
3. Initialisieren (Aufruf einer Adresse, meist das erste Byte)
4. Den überflüssigen Speicherplatz reklamieren (HIMEM wieder etwas hochsetzen)

Um den verfügbaren Speicherplatz im Computer optimal zu nutzen, muß man für jedes Hilfsprogramm dessen Länge mit und ohne Initialisierungsroutine kennen. Trotzdem ist es unsicher, ob der Platz ausreichen wird.

Das folgende BASIC-Programm erleichtert die ganze Sache aber erheblich. Wenn man dieses Programm zusammen mit allen RSX-Erweiterungen auf eine Diskette abspeichert, kann man alle gewünschten Routinen menügesteuert laden und initialisieren lassen.

```
100 MODE 1
110 PRINT
120 PRINT "************ RSX-BINDER ************
130 PRINT
140 PRINT "       (c) G.Woigk vs.: 22.4.86
150 PRINT
160 '
170 WINDOW 1,40,6,25
180 CLOSEIN:CLOSEOUT:SYMBOL AFTER 256:adr=HIMEM+1:MEMORY 10000
190 '
200 ' Arbeite alle Datazeilen ab:
210 '
220 ON ERROR GOTO 610           ' -> DATA exhausted -> fertig
230   WHILE 1
240   READ n$,anfang,start,ende
250   PRINT adr;">>> ";
260   GOSUB 310:IF a$="J" THEN GOSUB 390
270   WEND
280 '
290 ' Eingabe von 'J' oder 'N'
300 '
310 PRINT LEFT$((n$+"        "),8);" (J/N) ? ";
320 a$=""
330   WHILE INSTR(" JN",a$)<2
340   a$=UPPER$(INKEY$)
350   WEND
360 PRINT a$
370 RETURN
380 '
390 ' Initialisieren einer Routine
400 '
410 '    n$ = Name der Routine
420 'anfang = Adresse des 1. Bytes      (laut ORG-Anweisung im Assembler)
430 '  ende = Adresse des letzten Bytes + 1
```

```
440 ' start = Adresse des ersten Bytes, das nach der Initialisierung
450 '         der Routine bestehen bleiben muss
460 '
470 IF adr+start-ende<&4000 THEN PRINT"** Speicher voll! **";CHR$(7) :
RETURN
475 '
480 adr=adr+anfang-ende    ' Speicher reservieren
490 LOAD n$,adr            ' Mcode laden
500 CALL adr               ' Initialisieren
510 adr=adr+start-anfang   ' Anteil der init-Routine wieder vergessen
520 RETURN
530 '
540 '     Progname.Ext   Anfang   Start   Ende
545 '
550 DATA "LINECOPY.BIN" , &9c40 , &9cdd , &9f14
560 DATA "GRACOM  .BIN" , &7530 , &75DF , &786D
570 DATA "RIBBON  .BIN" , &7530 , &765e , &79fc
580 DATA "SQR     .BIN" , &7530 , &757e , &75d9
590 '
600 '
610 RESUME 620
620 MEMORY adr-1
630 NEW
```

Z80-RELOCALISATOR

Der Schneider CPC unterstützt sehr weitreichend den Wunsch, Zusatzroutinen nach Bedarf zu einem Hauptprogramm dazuladen zu können. Dabei ist es prinzipiell nicht vermeidbar, daß der (noch) freie Speicher dynamisch an die verschiedenen Erweiterungsprogramme verteilt wird.

Ein Programmpaket mit Systemerweiterungen kann also nicht damit rechnen, an eine bestimmte feste Adresse geladen zu werden!

Ideal wäre deshalb ein Maschinencode-Programm, das vollkommen ortsunabhängig geschrieben wurde. Das ist beim Z80 leider nur in ganz wenigen Fällen, vor allem bei ganz primitiven, kurzen Routinen möglich, da der Z80 fast nur über Befehle mit absoluter Adressierung verfügt.

Vorausgesetzt, die Ladeadresse ist zwar zunächst einmal unbekannt, die Lage des Programms ändert sich nach der Initialisierung aber nicht mehr (wie das in Strings der Fall sein kann), so genügt es, bei der Initialisierung des Programmpaketes alle absoluten Adressen an die tatsächliche Position des Programms anzupassen.

Adressierungen relativ zum Programmzeiger (JR und DJNZ) müssen dagegen nicht angepaßt werden. Entfernungen innerhalb eines Programms ändern sich ja nicht, wenn es an eine anderen Adresse geladen wird.

Funktionsweise

Dieser einfache Z80-Relocalisator benötigt für seine Arbeit eine Tabelle, in der man alle Stellen eintragen muß, an denen eine absolute Adresse verwendet wurde.

Er wird bei der Programmentwicklung vor das eigentliche Programm gehängt und mit diesem zusammen abgespeichert. Nachher lädt man das Programm an eine beliebige Stelle und ruft den Relocalisator auf, der dann alle absoluten Adressen anpaßt.

Zunächst einmal muß er feststellen, wo das Programm im Speicher überhaupt liegt. Dazu wartet er einfach auf den nächsten Interrupt (HALT), der als Unterprogrammaufruf die RETURN-Adresse auf den Maschinenstapel PUSHt. Das ist die mit dem Label "STELLE" markierte Stelle.

Mit dem RETURN vom Interrupt wird dieser Stack-Eintrag aber wieder verbraucht. Deshalb wird der Stapelzeiger wieder um zwei Stellen vermindert (DEC SP), um mit POP HL die Rücksprungadresse ein zweites Mal vom Stack holen zu können.

Sollte aber ausgerechnet zwischen den beiden DEC SP-Befehlen erneut ein Interrupt auftreten, wäre ein Byte der Rücksprungadresse zerstört. Das ist normalerweise aber fast ausgeschlossen, weil ja gerade erst ein Interrupt bearbeitet wurde. Der nächste Interrupt dürfte erst wieder nach einer knappen 300stel Sekunde auftreten. Wer ganz sicher gehen will, kann die beiden DEC SP-Befehle noch mit DI und EI rahmen, wobei das Label STELLE dann auf den Befehl DI zeigen muß.

Danach wird aus der tatsächlichen Lage von STELLE (im HL-Register) und der nominalen Lage (im DE-Register) die Verschiebeweite errechnet. Dieser Wert kann, je nach Verschieberichtung, positiv oder negativ sein und wird für den Rest der Verschiebung im Doppelregister BC gespeichert.

Nun kann zu jeder absoluten Nominaladresse deren augenblicklicher Wert berechnet werden, indem einfach der Offset aus dem BC-Register addiert wird.

Die erste absolute Adresse, die angepaßt werden muß, ist die Adresse der Tabelle selbst, damit der Relocalisator sie überhaupt findet.

In der Programmschleife ab RELOOP werden dann alle absoluten Adressen im eigentlichen Programm angepaßt. Bei jedem Schleifendurchgang ist zunächst einmal im BC-Register der Offset und im HL-Register der Zeiger in die Tabelle enthalten.

In der Tabelle stehen dabei die "unverschobenen absoluten Adressen der Stellen, an denen eine absolute Adresse steht, die verschoben werden muß." Das muß man sich klarmachen, um zu verstehen, was nun in jedem Schleifendurchgang gemacht werden muß.

Aus einem bestimmten Grund, der nachher noch erklärt wird, werden in die Tabelle dabei nicht die Adresse der Adresse selbst, sondern die Adresse der Stelle davor eingetragen.

Zunächst wird eine Adresse aus der Tabelle in das DE-Register gelesen und der Tabellenzeiger HL um zwei Bytes weitergestellt.

Dann wird erst einmal auf das Tabellenende getestet, wofür der Eintrag &0000 vorgesehen ist.

Danach werden das HL- und DE-Register vertauscht, weil nur im HL-Register gerechnet werden kann.

Zunächst wird zur Nominaladresse aus der Tabelle der Offset BC addiert, wodurch man die tatsächliche Adresse der zu verschiebenden Adresse erhält. Danach kann die Adresse endlich selbst angepaßt werden, was über zwei Byte-Additionen im Akku geschieht.

Zum Schluß wird der Tabellenzeiger, der die ganze Zeit im DE-Register unverändert blieb, wieder nach HL zurückgetauscht. Auch der Offset im BC-Register wurde nicht angetastet, so daß man jetzt wieder zum Schleifenkopf springen kann.

Programmentwicklung

Zunächst wird ein neues Programm ganz normal für eine feste Startadresse geschrieben und dort ausgetestet, bis es fehlerfrei läuft.

Danach muß man den Relocalisator *vor* den Programmtext hängen. (Nur dann kann man nach der Initialisierung den Relocator und seine Tabelle "vergessen" und den Platz wieder für andere Aufgaben benutzen.) Wie das im einzelnen geschieht, ist von Assembler zu Assembler unterschiedlich. Bei einigen kann man mehrere Source-Dateien "linken" (logisch verbinden, so daß sie zusammen einen "Gesamt-Quellcode" bilden). Im Zweifelsfall muß man (evtl. mit Hilfe eines Textverarbeitungsprogramms) aus der Textdatei mit dem Programm und der mit dem Relocator eine einzige machen.

Zum Schluß muß man auch noch die Tabelle eingeben. Das ist der langweiligste Teil des Ganzen. Trotzdem muß man hier äußerst sorgfältig vorgehen.

Dafür durchsucht man das ganze Programm systematisch nach allen Stellen, an denen eine absolute Adresse vorkommt. Das sind zum Beispiel folgende Befehle:

```
CALL adresse
JP   adresse
LD   HL,(adresse)
LD   (adresse),DE
```

Einige Befehle *können* eine absolute Adresse enthalten, was im Einzelfall aus dem Programmzusammenhang hervorgehen muß:

```
LD   HL,adresse
DEFW adresse
```

Jede gefundene absolute Adresse wird dann darauf untersucht, ob eine Stelle innerhalb des Programms adressiert wird:

- Liegt die Adresse außerhalb des Programms, und kann sie sich überhaupt nicht ändern, wenn das Programm verschoben wird, dann darf sie natürlich auch nicht verschoben werden. Hierbei handelt es sich in der Regel um Systemvariablen und Unterprogramme des Betriebssystems, die normalerweise über die entsprechenden Vektoren der Firmware-Jumpblocks angesprungen werden sollten.

- Liegt die Adresse aber im Programm, so muß diese Stelle in die Tabelle eingetragen werden.

Markieren von absoluten Adressen

Dazu markiert man die Stelle, an der die Adresse steht (nicht die adressierte Stelle!) mit einem Label und trägt dieses Label in die Tabelle ein.

Da es sich in den meisten Fällen um einen 3-Byte-Befehl handelt, bei dem der Adresse ein BefehlsByte vorangeht, ist es sinnvoll, nicht die Stelle direkt zu markieren, sondern die Adresse des Bytes davor in die Tabelle einzutragen. Das bedeutet nämlich, daß man das Label dann vor das BefehlsByte und im Assembler-Quelltext also direkt vor den gesamten Befehl setzen kann.

Bei 4-Byte-Befehlen und bei Adressen in DATA-Bereichen (mittels DEFW) geht das nicht. Hier muß man "die Stelle davor" mit der Assembler-Pseudo-Operation EQU markieren.

Um den Überblick zu behalten, sollte man für die Relocator-Label eine einheitliche Nomenklatur verwenden, um sie sofort von den "normalen" Labels unterscheiden zu können. Die in diesem Beispiel verwendete Version mit fortlaufenden Nummern ist dabei empfehlenswert, weil dann auch kein Label in der Tabelle vergessen wird.

Außerdem sollte man sich nicht verlocken lassen, ein "normales" Label mitzubenutzen, weil es schon an der richtigen Stelle steht. In diesem Fall versehen Sie lieber eine Stelle im Programm mit zwei Labels, dem "normalen" und dem für den Relocalisator. Sonst geht die Übersicht schnell verloren!

Sind dann alle Stellen markiert und in die Tabelle eingetragen, geht es ans letzte Austesten:

Läuft das Programm nicht, wenn man es an andere Stellen lädt, kann es daran liegen, daß eine Stelle vergessen oder auch zuviel markiert wurde. Oder man hat einmal daneben gegriffen und z.B. LD BC,(xxxx), einen 4-Byte-Befehl, direkt und nicht mit EQU markiert.

Markieren absoluter Adressen für den Relocalisator:

2-Byte-Befehl:

```
X1:   EQU  $-1
      DEFW adresse
```

3-Byte-Befehle:

```
X2:   LD   HL,(adresse)
X3:   LD   (adresse),HL
X4:   LD   BC,adresse
X5:   LD   (adresse),A
X6:   CALL adresse
X7:   JP   adresse
```

4-Byte-Befehle:

```
X8:   EQU  $+1
      LD   BC,(adresse)
X9:   EQU  $+1
      LD   (adresse),DE
X10:  EQU  $+1
      LD   IX,adresse
X11:  EQU  $+1
      LD   (adresse),IY
```

Achtung: Es gibt auch einen 4-Byte-Befehl: LD HL,(adresse). Praktisch alle Assembler generieren hierfür aber den entsprechenden 3-Byte-Befehl.

Mit diesem Relocator können fast alle Z80-Programme verschoben werden. Ausgenommen sind nur solche Konstruktionen, bei denen eine Adresse aus

zwei Bytes zusammengesetzt wird. Das kommt aber nur ausgesprochen selten vor, und meist läßt sich ein solches Programm dann sowieso nur noch in 256-Byte-Schritten verschieben.

```
; ****************************************
; Relocator und RSX-Binder fuer RSX-Routinen   (c) G.Woigk vs.: 22.4.86
; ****************************************
;
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;    Relocalisiere das Programm
; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
;
        ORG 30000

; Bestimme den Adress-Offset des Programms zwischen der momentanen realen
; Lage des Programms und dessen unverschobener Nominallage (laut
; ORG-Anweisung):
;
RELOC:  EI              ; Warte auf den naechsten Interrupt
        HALT            ; Dieser Pusht als Unterprogrammaufruf die
;                       ; RETURN-Adresse = 'STELLE' auf den Stack.
;
STELLE: DEC  SP         ; Hole RETURN-Adresse vom Stack:
        DEC  SP
        POP  HL         ; HL = reale Adresse von 'STELLE'
;
        LD   DE,STELLE  ; DE = unverschobene Adresse von 'STELLE'
        AND  A
        SBC  HL,DE      ; HL = Offset
        LD   B,H
        LD   C,L        ; BC = Offset
;
; Bestimme Lage der Tabelle mit den zu verschiebenden Adressen:
;
        LD   HL,RTABEL  ; HL = Unverschobene Adresse der Tabelle
        ADD  HL,BC      ; HL = Reale Adresse der Tabelle
;
; Nun Schleife ueber alle Adressen. Das Tabellenende ist mit #0000
; markiert:
;     BC enthaelt die ganze Zeit den Adress-Offset.
;     HL dient als Zeiger auf die Adressen in der Tabelle.
;
RELOOP: LD   E,(HL)     ; Hole eine Adresse aus der Tabelle
        INC  HL
        LD   D,(HL)     ; DE = unverschobene Adresse
        INC  HL         ;      einer unverschobenen Adresse
;
        LD   A,E        ; Test auf Tabellenende
        OR   D
X0:     JP   Z,BIND     ; Ende => RSX einbinden, weitere
                        ; Initialisierungen usw.
;
        EX   DE,HL      ; DE := Tabellenpointer
```

```
;                       ; HL := unversch. Adresse der unversch. Adresse
        ADD  HL,BC      ; HL := reale Adresse der unverschobenen Adresse
;
; Nun die Adresse (Lage: ab HL+1 !) anpassen:
;
        INC  HL         ; Zuerst das niederwertige Adressbyte
        LD   A,(HL)
        ADD  A,C
        LD   (HL),A
;
        INC  HL         ; Dann das hoeherwertige Adressbyte
        LD   A,(HL)
        ADC  A,B        ; (Addition mit ADC, um evtl. uebertrag
                        ; mitzunehmen)
        LD   (HL),A
;
        EX   DE,HL      ; HL := Tabellenpointer
        JR   RELOOP     ; und weiter zur naechsten Adresse
;
;
;       Adressen - 1 aller zu verschiebenden Worte!
;       -------------------------------------------
RTABEL: DEFW X0,X1,X2,X3,X4,X5
        DEFW #0000      ; Endmarke der Tabelle
;       -------------------------------------------
;
;
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;    Einbinden der RSX-Sammlung
; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
;
X1:
BIND:   LD   BC,ITABEL  ; RSX-Sammlung dem Kernel vorstellen
X2:     LD   HL,ISPACE
        CALL #BCD1      ; KL LOG EXT
;
        RET             ; Fertig; eventuelle, weitere
                        ; Initialisierungsroutinen
;                       ; moeglichst hier einfuegen, da der Platz bis zur
;                       ; Tabelle ISPACE  wieder freigegeben werden kann.
;
;
;////////////////////////////////////////////////////////////////////////
; Bis hierhin kann das Programmpaket mit Relocator, RSX-Binder und
; weiteren Initialisierungen nach der Initialisierung wieder
; ueberschrieben werden !
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;
ISPACE: DEFS 4          ; Raum fuer den Kernel (fuer eine verkettete
                        ; Liste aller RSX-Erweiterungen im RAM)
;
;       Tabelle der Routinen-Namen
;       Letzter Buchstabe jeweils mit #80 geodert
;       -----------------------------------------
```

```
NAMTAB: DEFM "HARDCOP"
        DEFB "Y"+128
        DEFM "TEXTCOP"
        DEFB "Y"+128
;       ----------------------------------------
        DEFB 0      ; Endemarke der Tabelle
;
;
X3:     EQU  $-1
ITABEL: DEFW NAMTAB
;
;       Tabelle mit Jumps zu den Routinen (je 3 Bytes)
;       ----------------------------------------------
X4:     JP   HARD       ; fuer "HARDCOPY"
X5:     JP   TEXT       ; fuer "TEXTCOPY"
;       ----------------------------------------------
;
; Ab hier folgen die Maschinencode-Routinen
;
; ....
```

Die Sprungleisten des Betriebssystems

Neben der Hardware gehört bei jedem Computer immer auch Software zum Lieferumfang. Am wichtigsten ist hierbei die Unterscheidung in Betriebssystem und das erwünschte Programm.

Hinzu kommt beim Schneider CPC noch die fest vorgesehene Möglichkeit für Systemerweiterungen:

- Betriebssystem (auch: OS – Operating System)
- Hintergrundprogramme (Systemerweiterungen)
- Vordergrundprogramm (laufendes Programm)

Im Falle eines Interpreters als Vordergrundprogramm (beispielsweise BASIC) kommt noch eine Feinheit hinzu, nämlich daß der Interpreter eine Programmtext-Datei "abarbeiten" kann.

Im Betriebssystem sind eine Vielzahl von Routinen zusammengefaßt, die von allen Programmen benötigt werden oder auch nur benötigt werden könnten. Dabei handelt es sich fast nur um Ein-/Ausgaberoutinen.

Beim Schneider CPC mit seiner komplizierten Speicherverwaltung kommen auch noch hierfür Routinen hinzu. Auch für die Programm-/Ablaufsteuerung sind im Betriebssystem Routinen enthalten.

Beim Schneider CPC sind alle Routinen sehr sorgfältig nach Funktionen gegliedert worden:

- Kernel (Kern, Zentrale):
  Restarts (Befehlserweiterungen für Speicherverwaltung)
  Speicherverwaltung
  Interrupt-Mechanismus
  Externe Kommandos, Hintergrund-ROMs
- Machine Pack (Hardware-nahe Routinen):
  Drucker-Port
  Start von Vordergrundprogrammen
  Programmierung der ULA, des CRTC und des PSG
- Key Manager (Tastaturroutinen)
- Text-VDU (Textausgabe auf dem Bildschirm)
- Grafik-VDU (Grafikausgabe auf dem Bildschirm)
- Screen Pack (Routinen für die Text- und Grafik-VDU)
- Cassette Manager (Kassetten-Interface)
- Sound-Manager (Musik- und Geräuschausgabe)

Wie man an dieser Aufstellung sieht, befassen sich wirklich fast alle Möglichkeiten mit Ein- oder Ausgabeschnittstellen. All diese Routinen sind im unteren ROM zusammengefaßt, das auf den Adressen &0000 bis &3FFF eingeblendet werden kann. In diesem ROM sind zusätzlich noch zwei Abteilungen untergebracht, die von AMSTRAD her nicht mehr zum Betriebssystem gezählt werden:

- Floating Point Pack (Fließkomma-Rechenroutinen)
- Editor (Zeileneditor, der von BASIC ständig benutzt wird)

Es ist äußerst sinnvoll, die zum Teil recht komplexen Routinen für die Ein- und Ausgabeverwaltung nur einmal zu programmieren und dann in einem Betriebssystem zusammenzufassen. Diese Routinen können dann von allen Hauptprogrammen benutzt werden. Die Programmierer brauchen sich nicht mehr um Hardware-spezifische Eigenheiten des Computers zu kümmern.

Alle Routinen des Betriebssystems sind aber mit zwei kleinen Schönheitsfehlern behaftet, wovon der eine nur für den CPC typisch ist:

- Alle Routinen liegen im unteren ROM, das normalerweise immer ausgeblendet ist. Vor Aufruf einer Routine des Betriebssystems muß man also immer erst das untere ROM einblenden.

- Kaum ein Betriebssystem ist fehlerfrei, und man muß deshalb damit rechnen, daß in zukünftigen Versionen des Computers Änderungen am Betriebs-

system vorgenommen werden, wie das beim CPC 664 und CPC 6128 auch tatsächlich schon der Fall war. Die Einsprungadressen für die einzelnen Adressen können sich also ändern.

Es gibt aber eine Methode, mit der man beide Probleme in den Griff bekommen kann:

JUMPBLOCKS – SPRUNGLEISTEN

Zunächst einmal werden die Einsprung-Adressen für die wichtigsten Routinen vom Hersteller des Betriebssystems garantiert. Da das von der Programmierung her nur sehr schwer zu erreichen ist (Eine Änderung an einer Stelle im Programmcode verschiebt zwangsläufig alle nachfolgenden Routinen im ROM), faßt man die Einsprungadressen zu sogenannten "Sprungleisten" zusammen: Ein bestimmter Bereich im Speicher besteht nur aus Sprungbefehlen zu den einzelnen Betriebssystem-Routinen. Ein Eintrag in diesem "Jumpblock", also ein 3-Byte-Z80-Sprungbefehl, wird als Vektor bezeichnet.

Werden jetzt Änderungen im Programmcode vorgenommen, so verschieben sich zwar die Anfangsadressen der einzelnen Routinen. Die Lage der Sprungleiste und also auch die Lage aller Vektoren bleibt aber gleich. In jedem Vektor ist jetzt nur ein Sprung zu der geänderten Routinenadresse eingetragen.

Programme, die nur die Vektoren und nicht direkt die Betriebssystem-Routinen aufrufen, haben also gute Aussichten, auch bei einem modifizierten Betriebssystem ohne Änderungen lauffähig zu bleiben.

Damit wäre also das Problem der festen Einsprungadressen gelöst. Wie steht es aber mit der Tatsache, daß man vor jedem Aufruf einer Betriebssystem-Routine das untere ROM einblenden muß?

BEFEHLSERWEITERUNG MIT RESTART

Die Software-Entwickler bei AMSTRAD haben dies Problem mit dem LOW KERNEL JUMPBLOCK, einer Sprungleiste, die im Bereich der Z80-Restarts liegt gelöst: von &0000 bis &003F.

Mit den Vektoren in dieser Sprungleiste können Unterprogramme aufgerufen (CALL) oder angesprungen (JP) werden, wobei der ROM-Status oder auch die gesamte ROM-Konfiguration für die Zeit der Unterprogramm-Bearbeitung wählbar ist.

Das heißt, man ruft einen Vektor im LOW KERNEL JUMPBLOCK auf, übergibt ihm auf die jeweils vorgeschriebene Weise die Adresse der Routine, die man aufzurufen gedenkt, und Informationen über den von der Routine benötigten ROM-Status bzw. die ROM-Konfiguration. Die Behandlungsroutine des Vektors stellt die ROM-Konfiguration ein, ruft das gewünschte Unterprogramm auf und restauriert abschließend sogar wieder die alte ROM-Konfiguration.

Damit ist es möglich, bei eingeschaltetem RAM eine Routine in einem ROM aufzurufen, weil man vom kurzzeitigen Einblenden eines ROMs nichts mitbekommt.

Da der Aufruf eines Vektors des LOW KERNEL JUMPBLOCKs meist geschieht, wenn unten kein ROM eingeblendet ist, müssen die Vektoren nicht nur im ROM, sondern auch im RAM auf den gleichen Adressen eingetragen sein. Sie werden deshalb bei der Initialisierung des Rechners aus dem ROM ins RAM kopiert.

Daß man für diese Aufgaben den Bereich der Z80-Restarts gewählt hat, ist nicht ohne Hintergedanken geschehen. Die Restarts sind ja Unterprogrammaufrufe, die im Unterschied zum normalen CALL-Befehl nur ein und nicht drei Bytes beanspruchen.

Das hat man sich zunutze gemacht und die "Parameter"-Übergabe, also die Angabe von Routinenadresse und ROM-Konfiguration so festgelegt, daß man mit den Restarts praktisch neue Befehle geschaffen hat. Bei einem normalen Z80-CALL-oder -JP-Befehl folgen auf das Befehlsbyte (205 oder 195) als "Argument" zwei Bytes mit der Sprungadresse.

Die Restarts sind ebenfalls ein Byte lang und erwarten u.a. als Argument die Sprungadresse. Was liegt also näher, als diese Adresse in gleicher Weise an das Befehlsbyte anzuhängen? In einem Assembler-Quelltext sähe das dann so aus:

```
RST  #08
DEFW adresse
```

Damit hat man einen neuen Befehl erfunden, der CALL oder JP ersetzen kann. Zu lösen ist nur noch die Frage, wie man auch noch die ROM-Informationen übergibt. Beim Restart 1 (RST #08) ist das folgendermaßen geregelt.

Aufgerufen werden können mit diesem erweiterten Sprungbefehl (JP) nur Routinen im unteren ROM oder RAM. Das heißt, die Adresse liegt immer im Bereich &0000 bis &3FFF, und die obersten beiden Adreßbits A14 und A15 sind immer Null.

Das macht man sich zunutze, um nun hier den gewünschten ROM-Status zu übergeben: Ein gesetztes Bit 14 blendet das untere ROM aus, und ein gesetztes Bit 15 blendet das obere ROM aus (nur für die Zeit des Unterprogramms). Es ergeben sich folgende vier Kombinationen:

```
A14  A15 ROM-Status:
-------------------------------------------------------------------------------
 0    0  oben ROM. unten ROM  Aufruf einer Routine im Betriebssystem-ROM
 0    1  oben RAM. unten ROM  Aufruf einer Routine im Betriebssystem-ROM
 1    0  oben ROM. unten RAM  Aufruf einer Routine im RAM
 1    1  oben RAM. unten RAM  Aufruf einer Routine im RAM
```

Ein Sprung zu einer Routine des Betriebssystems ab der Adresse &1234 mit eingeblendetem Bildschirm-RAM im obersten Adreßviertel würde dann wie folgt erreicht:

```
RST  #08
DEFW #1234 + #8000
```

Zur Adresse wird noch &8000 addiert, wodurch das Bit A15 gesetzt ist, das das obere ROM aus- und das Bildschirm-RAM einblendet. Das Betriebssystem-ROM wird nicht aus-, sondern eingeschaltet, weil Bit A14 nicht gesetzt ist (sonst hätte man auch noch &4000 addieren müssen).

Noch ein weiterer Punkt war zu beachten, um diese Vektoren einem Z80-Befehl so ähnlich wie nur irgend möglich zu machen: Es durften keine Register verändert werden, weder beim Sprung zur Routine noch beim Rücksprung.

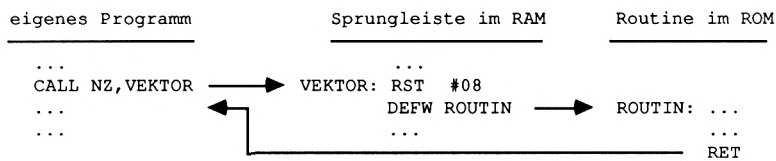
Alle notwendigen Berechnungen werden deshalb mit dem zweiten Registersatz der CPU durchgeführt, den man normalerweise nicht benutzen darf. Daraus ergibt sich eine weitere Eigenart aller Restart-Befehle: Alle Restarts lassen einen Interrupt wieder zu!

Das liegt daran, daß vor dem Registertausch der Interrupt verboten werden muß, da der Interrupt ebenfalls auf den zweiten Registersatz zugreifen will. Nachdem alle Berechnungen erledigt sind, werden die normalen Register wieder "zurückgeklappt" und ein Interrupt wieder zugelassen.

Nun könnte man also eine Sprungleiste im unteren ROM einrichten, auf die man mit erweiterten CALL-Befehlen zugreifen kann. Daß man im Schneider CPC aber speziell für Aufrufe von Routinen im untersten Adreßviertel keine CALL-, sondern nur zwei JP-Ersatzroutinen mit den Restarts 2 und 5 gebildet hat, zeigt, daß man bei AMSTRAD noch einen anderen Weg gegangen ist: Die Sprungleiste für die Betriebssystem-Routinen werden bei der Initialisierung des Rechners in das zentrale RAM kopiert. Die Vektoren selbst werden mit den erweiterten JP-Befehlen gebildet.

Man ruft also nicht mit einem Restart-CALL einen normal mit JP gebildeten Vektor im ROM auf (der dann endgültig die Betriebssystem-Routine anspringt), sondern ruft mit einem normalen CALL den Vektor im RAM auf, der mit einem erweiterten JP-Befehl gebildet ist.

Das hat zwei Vorteile: Zum einen kann man auch weiterhin bedingte Aufrufe von Unterprogrammen benutzen. Wäre bereits der Aufruf des Vektors mit einem CALL-Restart gebildet worden, so müßte man für jede Aufrufbedingung einen anderen Restart verbrauchen:



Der andere Vorteil ist, daß man alles im RAM ändern kann. Man kann in einen Vektor also einen Sprung zu einer anderen eigenen Routine eintragen, die dieselbe Aufgabe wahrnimmt, nur eben besser, schneller, fehlerfrei oder leicht verändert. Dieses "Umbiegen" wird "Patchen" genannt (patch = flicken).

PATCHEN VON VEKTOREN

Das Patchen muß dabei so unauffällig wie möglich vor sich gehen. Auf die Vektoren greift beim Schneider CPC ja nicht nur ein einziges Programm zu. Hier können gleichzeitig ein Vordergrund- und beliebig viele Hintergrundprogramme im Speicher vorhanden sein, die alle Routinen des Betriebssystems benutzen können.

Änderungen, die man für das eine Programm vorgesehen hat, können katastrophale Folgen für ein anderes Programm nach sich ziehen, wenn dieses den veränderten Vektor benutzen will. Die ursprüngliche "Modul"-Beschreibung des gepatchten Vektors darf möglichst nicht geändert werden:

Eingabe – Funktion – Ausgabe

Zur Ein- und Ausgabe gehören dabei die Beschreibung der Registerschnittstellen:

- In welchen Registern werden welche Eingaben erwartet?
- In welchen Registern werden welche Ausgaben gemacht?
- Welche Register bleiben garantiert unverändert?

Auch die Funktion muß möglichst die gleiche bleiben. Trotzdem ist es aber immer gerade die Funktion eines Vektors, die durch einen Patch geändert werden soll. Die Änderung muß sich aber in Grenzen halten.

So ist es beispielsweise eine beliebte Idee, die Textausgabe vom Bildschirm auf den Drucker umzulenken, um beispielsweise auch einmal einen Diskettenkatalog zu Papier bringen zu können.

Die bekanntesten Patches werden wohl von AMSDOS vorgenommen. Viele Vektoren zum Cassette Manager werden "umgebogen", um Ein- und Ausgabedateien nun auf einem Diskettenlaufwerk zu verwalten.

Beim Patchen sind dabei folgende Punkte zu beachten:

1. Es dürfen keine zusätzlichen Eingaben erwartet werden. Man kann allerdings weniger verlangen als der Originaleintrag.

2. Alle Ausgaben, die vom Originaleintrag gemacht wurden, müssen auch vom Patch an das aufrufende Programm zurückgeliefert werden.

3. Alle Register, die bisher unverändert blieben, dürfen auch durch die neue Routine nicht verändert werden.

4. Es sollten keine völlig sinnentstellenden Änderungen an einzelnen Vektoren vorgenommen werden.

Darüber hinaus gibt es noch einige praktische Erwägungen. So ist es oft sinnvoll, den alten Eintrag zu retten.

Dann kann man eine Änderung zu einem späteren Zeitpunkt auch wieder rückgängig machen, oder man kann die Originalroutine noch mitbenutzen, nachdem man beispielsweise einige Vorarbeiten oder zusätzliche Tests vorgenommen hat.

5. Meist ist es sinnvoll, den alten Vektor zu retten, bevor man seinen Patch installiert.

Deshalb müssen auch alle Vektoren ortsunabhängig sein. Deshalb handelt es sich um einen normalen Sprungbefehl. Relative Sprünge in Vektoren sind nicht sehr ratsam.

6. Der Patch muß ortsunabhängig (position independant) sein.

Achtung: Die von AMSDOS gepatchten Vektoren sind nicht ortsunabhängig!

Wer die gepatchten Vektoren von AMSDOS erneut patchen will, muß einigen Umstand in Kauf nehmen, wenn er die "Original"-AMSDOS-Vektoren noch mitbenutzen will. Das muß dann etwa so vor sich gehen:

- Initialisieren: alten (AMSDOS-) Vektor retten, eigenen Vektor eintragen.

- Wenn jetzt die eigene Routine aufgerufen wurde:
  evtl. eigenen Programmcode abarbeiten.
  Alten Vektor wieder restaurieren und den Vektor aufrufen (Originalfunktion).
  Danach wieder den eigenen Vektor eintragen.
  Evtl. eigenen Programmcode abarbeiten und RETURN.

Es ist nicht gerade vorbildlich, wie man sich hier bei AMSTRAD über die selbst erfundenen Spielregeln hinwegsetzte.

Katalog auf dem Drucker

Das folgende Beispielprogramm erstellt einen Diskettenkatalog auf dem Drukker. Das Ganze ist als eine RSX-Erweiterung eingebunden und wird in BASIC mit "|LCAT" aufgerufen.

Die zugehörige Routine patcht zunächst den Textausgabevektor TXT OUTPUT, so daß Ausgaben für den Bildschirm zum Drucker umgeleitet werden. Danach wird ein Katalog angefordert, wofür der Vektor CAS CAT aufgerufen wird.

Die AMSDOS-Routine, die den Katalog ausgibt, ruft den Vektor TXT OUTPUT auf, um so alle Zeichen zum Bildschirm zu schicken. Da dieser Vektor aber gerade vorher gepatcht wurde, landen die Zeichen wie gewünscht auf dem Drucker.

Nachdem der Katalog ausgegeben wurde, wird der alte Vektor in TXT OUTPUT wieder restauriert.

Dieses Programm ist jedoch nicht geeignet, auch einen Kassettenkatalog zu erstellen. Die Katalogroutine des Cassette Managers ruft nämlich, anders als AMSDOS, nicht den Vektor für TXT OUTPUT, sondern diese Routine direkt auf.

Das Assemblerprogramm ist auch gleich mit dem Z80-Relocator zusammengebunden, so daß man sich hier noch einmal seine Funktionsweise anschauen kann.

```
; RSX-Erweiterung fuer einen Diskettenkatalog auf dem Drucker
; ----------------------------------------------------------
;
        ORG 30000
;
; 0. Label-Deklarationen
;
LOGEXT: EQU  #BCD1      ; KL LOG EXT
CASCAT: EQU  #BC9B      ; CAS CAT
TXTOUT: EQU  #BB5A      ; TXT OUTPUT
MCPRNT: EQU  #BD2B      ; MC PRINT CHAR
;
; 1. Relocalisiere das Programm:
;
RELOC:  EI
        HALT
STELLE: DEC  SP
        DEC  SP
        POP  HL
        LD   DE,STELLE
        AND  A
        SBC  HL,DE
        LD   B,H
        LD   C,L
        LD   HL,RTABEL
        ADD  HL,BC
RELOOP: LD   E,(HL)
        INC  HL
        LD   D,(HL)
        INC  HL
        LD   A,E
        OR   D
X0:     JP   Z,BIND
        EX   DE,HL
        ADD  HL,BC
        INC  HL
        LD   A,(HL)
        ADD  A,C
        LD   (HL),A
        INC  HL
        LD   A,(HL)
        ADC  A,B
        LD   (HL),A
        EX   DE,HL
        JR   RELOOP
;
;       Tabelle aller zu relocalisierenden Stellen:
;       -------------------------------------------
RTABEL: DEFW X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10
        DEFW #0000          ; Endmarke der Tabelle
;       -------------------------------------------
;
; 2. Einbinden der RSX-Sammlung:
;
X1:
```

```
BIND:   LD   BC,ITABEL
X2:     LD   HL,ISPACE
        CALL LOGEXT
        LD   A,201
X3:     LD   (RELOC),A   ; sukzessive Initialisierungen verhindern
        RET
;
ISPACE: DEFS 4
;
;       Tabelle der Routinennamen.
;       --------------------------
NAMTAB: DEFM "LCA"
        DEFB "T"+#80
        DEFB 0
;
X4:     EQU  $-1
ITABEL: DEFW NAMTAB         ; Zeiger auf Namenstabelle
;
;       Tabelle mit Jumps zu den Routinen (je 3 Bytes)
;       ----------------------------------------------
X5:     JP   LCAT
;
; 3. RSX-Behandlungsroutine:
;
LCAT:   LD   HL,TXTOUT      ; Kopiere Originaleintrag des
X6:     LD   DE,PUFFER      ; Textausgabevektors TXT OUTPUT
        LD   BC,3           ; in einen Puffer
        LDIR
;
X7:     LD   HL,PATCH       ; Installiere Patch:
        LD   DE,TXTOUT      ; Kopiere Sprung zur eigenen
        LD   BC,3           ; Textausgaberoutine
        LDIR                ; in den Vektor
;
X8:     LD   DE,CASPUF      ; CAS CAT benoetigt 2 kByte Puffer.
        CALL CASCAT         ; Katalog ausgeben
;
X9:     LD   HL,PUFFER      ; Originaleintrag in
        LD   DE,TXTOUT      ; TXT OUTPUT restaurieren
        LD   BC,3
        LDIR
        RET                 ; fertig
;
PUFFER: DEFS 3              ; Speicher fuer Original-TXT-OUTPUT-Vektor
;
X10:
PATCH:  JP   DRUOUT         ; Ersatzsprung fuer TXT OUTPUT
;
; 4.Ersatz-Routine fuer TXT OUTPUT:
;
DRU1:   POP  AF
DRUOUT: PUSH AF             ; AF muss fuer TXT OUTPUT erhalten bleiben
        CALL MCPRNT         ; Versuche, Zeichen auszudrucken
        JR   NC,DRU1        ; Kein Erfolg: Try again
```

```
        POP  AF             ;           sonst fertig
        RET
;
CASPUF: DEFS #800
;
END
```

In der Ersatzroutine DRUOUT muß das AF-Doppelregister gerettet werden. Der Grund ist die Schnittstellenbeschreibung von TXT OUTPUT und MC PRINT CHAR.

Der Vektor TXT OUTPUT verändert keine Register, MC PRINT CHAR kann aber das A-Register verändern und setzt als Erfolgskontrolle das CY-Flag (oder auch nicht). Die Routine DRUOUT muß also einerseits das AF-Register retten, weil das für den Vektor TXT OUTPUT so verlangt wird, und andererseits, um das Zeichen, das ausgedruckt werden soll, wieder zu bestimmen, wenn der Ausdruck nicht gleich beim erstenmal klappte.

DIE SPRUNGLEISTEN

Im Schneider CPC gibt es insgesamt fünf Sprungleisten. Die Vektoren aller fünf Sprungleisten sind im Anhang mit ihrer vollständigen Modulbeschreibung aufgeführt.

Der erste ist der LOW KERNEL JUMPBLOCK mit den Restarts. Weil hier die Einträge in ROM und RAM gemacht sind, eignet er sich nicht besonders zum Patchen. Ausgenommen davon sind allerdings zwei Stellen: der Restart 6, dem vom Vordergrund-Programm eine beliebige Funktion zugeordnet werden kann, und der External Interrupt Entry.

Außerdem kann man den Restart 0 RESET ENTRY im RAM umwidmen. Der Software-RESET mit CTRL/SHIFT/ESC funktioniert dann auch weiterhin, weil er immer den ROM-Restart benutzt. Solange man also sicher ist, daß beim Aufruf des eigenen Restart 0 das untere ROM nie eingeblendet ist, kann nichts schiefgehen.

External Interrupt Entry

Der Kernel kann zwischen den normalen Ticker-Interrupts, die 300mal in jeder Sekunde von der ULA erzeugt werden, und Interrupt-Anforderungen von Erweiterungen am Systembus unterscheiden. Dazu dient die Dauer des In-

terrupt-Signals. Externe Interrupt-Quellen dürfen ihre Interrupt-Anforderung erst auf ausdrückliche Anweisung ihrer Behandlungsroutine zurücknehmen (oder frühestens ca. 20 Mikrosekunden nach Aufnahme der Interrupt-Behandlung).

Erkennt die Interrupt-Behandlungsroutine des Kernel, daß ein externer Interrupt vorliegt, so wird der Vektor auf der Adresse &003B aufgerufen.

Für jede Interrupt-erzeugende Systemerweiterung muß an dieser Stelle eine Behandlungsroutine eingeklinkt werden. Da man davon ausgehen muß, daß eventuell noch weitere solcher Erweiterungen angeschlossen sein könnten, muß der Vektor an dieser Stelle ordnungsgemäß gepatcht werden. Das heißt, man muß eine Kopie des alten Eintrags anlegen, die angesprungen werden muß, wenn man feststellt, daß der Interrupt doch nicht für die eigene Routine war.

Die Behandlungsroutine darf dabei die Register AF, BC, DE und HL verändern. Sie darf aber nicht den Interrupt wieder zulassen und auch die meisten Betriebssystem-Routinen nicht aufrufen. Am empfehlenswertesten ist es, den Event-Mechanismus des Kernel in Anspruch zu nehmen. Darauf wird im Kapitel über den Kernel noch einmal näher eingegangen.

Das folgende Assemblerprogramm veranschaulicht die allgemeine Vorgehensweise:

```
; Behandlungsroutine für eine Interrupt-erzeugende Hardware-Erweiterung
; ---------------------------------------------------------------------
;
INIT:  LD   HL,#003B  ; alten Eintrag an dieser Stelle retten
       LD   DE,KOPIE  ; Achtung: Hier stehen 5 Bytes fuer den Vektor
       LD   BC,5      ; zur Verfuegung, die evtl. auch ausgenutzt sein
       LDIR           ; koennten
;
       LD   HL,PATCH  ; eigenen Vektor an dieser Stelle installieren
       LD   DE,#003B
       LD   BC,3
       LDIR
;
       LD   A,#uv     ; Der Hardware mitteilen, dass sie
       LD   BC,#wxyz  ; ab sofort Interrupts erzeugen darf
       OUT  (C),A
       RET
;
KOPIE: DEFS 5         ; Platz fuer den alten Eintrag.
PATCH: JP   XTINT     ; Eigener Eintrag.
;
XTINT: LD   BC,#wxyz  ; Ueberpruefen, daß der Interrupt
       IN   A,(BC)    ; auch wirklich von der eigenen Erweiterung
       CP   #uv       ; ausgeloest wurde;
```

```
        JR    NZ,KOPIE  ; wenn nicht, muss die Kopie angesprungen werden
;
        LD    A,#uv     ; Interrupt-Signal abstellen
        LD    BC,#wxyz
        OUT   (C),A
        ....            ; Interrupt-Behandlung
        ....
        RET             ; fertig
```

User Restart

Restart 6 wird vom Betriebssystem nicht benutzt und kann vom Vordergrund-Programm nach Belieben eingesetzt werden. Da man Änderungen am Restart aber nur im RAM machen kann, ist der Restart im ROM so konstruiert worden, daß hier zunächst der aktuelle ROM-Status in der Speicherzelle &002B gerettet wird und danach der Restart im RAM wiederholt wird.

Als Beispiel folgt ein Assemblerprogramm, das vielleicht für CPC 6128-Programmierer ganz interessant ist. Ähnlich dem Restart 4 RAM LAM, mit dem man unabhängig vom aktuellen ROM-Status ein Byte aus dem RAM lesen kann, soll ein neuer Befehl gebildet werden, der das Byte immer aus der zusätzlichen RAM-Bank liest. 464- und 664-Benutzer können sich aber immerhin ansehen, wie dieser Restart eingesetzt werden kann:

```
; Restart 6: ZRAM LAM
;
; simuliert ein LD A,(HL) aus dem zusätzlichen RAM des CPC 6128
;
INIT:   LD    HL,PATCH        ; Initialisierung des Restart 6
        LD    DE,#0030
        LD    BC,3            ; Sprung zur eigenen Routine RAMLAM an die
        LDIR                  ; Adresse des Restart 6 kopieren.
;
        LD    A,#FF           ; #FF soll als Kennung dafuer dienen, dass
;       LD    (#002B),A       ; der ROM-Restart 6 in dieser Speicherzelle
;       RET                   ; keinen alten ROM-Status gespeichert hat;
;                             ; dass der Restart also bei
;                             ; eingeschaltetem unterem RAM
PATCH:  JP    RAMLAM          ; erfolgte
;
; Die Routine RAMLAM soll wie der Z80-Befehl LD A,(HL) funktionieren,
; das heisst, nur das A-Register wird verändert; F, BC, DE, HL, IX und
; IY bleiben unberührt:
;
RAMLAM: PUSH HL               ; HL retten
        PUSH AF               ; AF retten
;
        LD    A,H             ; Bits 14 und 15 der Adresse
;                             ; (Speicherviertel)
```

```
        RLCA                    ; in die Bits 0 und 1 von A schieben
        RLCA
        AND  7                  ; Bits 3 bis 7 nullen und
        OR   4                  ; Bit 2 setzen
        CALL #BD5B              ; KL RAM SELECT -> von &4000 bis &7FFF wird
;                               ; der entsprechende Block des zus. RAM
;                               ; eingeblendet
;
        RES  7,H                ; Adresse in HL in den Bereich &4000 bis
;                               ; &7FFF
        SET  6,H                ; zwingen
        LD   H,(HL)             ; und Wert zunächst nach H laden
;
        CALL #BD5B              ; KL RAM SELECT; A enthielt noch den alten
;                               ; RAM-Status, der hiermit restauriert wird
;
        LD   A,(#002B)          ; teste, ob RST 6 mit unterem ROM = Ein
;                               ; erfolgte
        CP   #FF
        CALL NZ,OLDROM          ; wenn ja, restauriere ROM-Status
;
        POP  AF                 ; restauriere F
        LD   A,H                ; Wert aus dem zus. RAM nach A laden
        POP  HL                 ; restauriere HL
        RET                     ; fertig
;
; Restauriere alten ROM-Status, der in &002B gespeichert war und jetzt
; im A-Register ist
;
OLDROM: DI                      ; zweiter Registersatz wird benutzt.
        EXX
;                               ; C' muss ständig korrekten ROM-Status (und
;                               ; Bildschirmmodus) enthalten; B' die
;                               ; I/O-Adresse
;                               ; der ULA
        LD   C,A                ; ---> Alte ROM-Konfiguration nach C' und
        OUT  (C),C              ;      die ULA damit programmieren
        EXX
        EI
        LD   A,#FF              ; in &002B wieder #FF eintragen, falls
;                               ; nächster
        LD   (#002B),A          ; Restart mit unten RAM = Ein erfolgt
        RET
```

HIGH KERNEL JUMPBLOCK

Außer dem LOW KERNEL JUMPBLOCK gibt es auch noch den HIGH KERNEL JUMPBLOCK im Bereich &B900 bis &B920. Hier finden sich nur Routinen, mit denen man ROMs umschalten kann, und außerdem LDDR- und LDIR-Routinen, die Speicherbereiche bei eingeblendeten RAMs verschieben. Letzteres sind für ROM-Software interessante Unterprogramme.

Ebenfalls zu diesem Jumpblock gehört noch &B921 HI KL POLL SYNCHRONOUS. Mit dieser Routine läßt sich sehr schnell testen, ob ein synchroner Interrupt mit einer höheren Priorität als der laufenden auf seine Ausführung wartet. Diese Routine ist aber nicht als Vektor ausgelegt, sondern beginnt direkt auf dieser Adresse.

Bei den CPC 664 und 6128 kommt dann noch mit der Adresse &B92A die Routine HI KL SCAN NEEDED dazu, die ebenfalls nicht als Vektor ausgeführt ist.

MAIN FIRMWARE JUMPBLOCK

Die eigentliche Sprungleiste zu den Betriebssystem-Routinen ist aber der MAIN FIRMWARE JUMPBLOCK, der auf der Adresse &BB00 beginnt und beim CPC 464 bis &BD39 reicht. Beim CPC 664 kommen noch einige Vektoren hinzu, vor allem für die Grafik-VDU, so daß dieser Jumpblock hier bis &BD5A geht. Beim CPC 6128 kommt auch noch der Vektor KL RAM SELECT dazu. Hier reicht der Jumpblock bis &BD5D.

In diesem Main Firmware Jumpblock befinden sich die (natürlich wieder mit Restarts gebildeten Vektoren) zu allen Abteilungen der Firmware. Dieser Jumpblock ist, wie sein Name auch schon sagt, für den Assemblerprogrammierer überhaupt der wichtigste von allen.

Der Jumpblock des BASIC-Interpreters

Nicht zum Betriebssystem zählt der Jumpblock, der vom Kernel für den BASIC-Interpreter eingerichtet wird. Trotzdem wird auch diese Sprungleiste durch die Betriebssystem-Routine JUMP RESTORE eingerichtet.

In diesem Jumpblock sind die Vektoren zum Zeileneditor, zu den Fließkomma-Routinen und (nur beim CPC 464) zu den Integer-Routinen zusammengefaßt.

Wünschenswert wäre gewesen, wenn auch dieser Bereich eine garantierte Lage im RAM hätte. Dem ist leider nicht so. Er wird immer direkt über den normalen Firmware-Jumpblock gelegt. Da dieser bei allen CPCs verschieden lang ist, ändert sich auch die Lage dieses Blocks. Zusätzlich hat man bei AMSTRAD ab dem CPC 664 die Integer-Routinen aus dem Betriebssystem ins BASIC-ROM verbannt, womit auch deren Vektoren wegfielen. Außerdem sind die verbliebenen Fließkomma-Vektoren umgestellt und auch hier ein paar gestrichen worden.

Wer also den Zeileneditor oder die Fließkomma-Routinen benutzen will, muß, sofern er Programme schreibt, die auf allen CPCs laufen sollen, hier ganz besonders aufpassen und zunächst einmal testen, auf welchem Rechner das Programm gerade läuft.

Die Indirections

Der letzte Jumpblock stellt eine weitere Besonderheit dar. Die Indirections sind eigentlich keine Vektoren, sondern Umleitungen von ROM-Routinen über das RAM.

Einige Routinen rufen an entscheidenden Stellen diesen "Vektor" im RAM auf, wo normalerweise nur wieder ein Sprung zurück zur Routine eingetragen ist. Der Umweg über das RAM bietet aber die Möglichkeit, hier eine eigene Routine zu installieren, um wie bei den normalen Vektoren die Funktion der einzelnen Betriebssystem-Routinen zu ändern.

Der Patch einer Indirection wirkt jedoch auf alle Programme im Computer. Auch Betriebssystem-Routinen, die andere Betriebssystem-Routinen aufrufen, werden davon beeinflußt, beim Patch eines Vektors jedoch nicht, weil die Betriebssystem-Routinen die Vektoren nicht selbst aufrufen.

Dabei sind die Indirections nur mit dem normalen Z80-Befehl JP gebildet, weil sie ja bereits vom unteren ROM aus angesprungen werden. Patcht man eine Indirection, so ist immer das untere ROM eingeblendet und der Status des oberen unbestimmt.

Außerdem stellen Patches einer Indirection meist einen Eingriff auf der untersten Ebene des Betriebssystems dar. Plausibilitätskontrollen, die viele Vektoren noch mit den ihnen übergebenen Argumenten vornehmen, sind hier nicht mehr üblich. Die Indirections liegen im Bereich von &BDCD bis &BDF3 (CPC 464) bzw. &BDF6 (CPC 664, 6128).

Die Abteilungen des Betriebssystems

Es folgen nun die Beschreibungen der einzelnen Firmware-Packs, wie sie im MAIN FIRMWARE JUMPBLOCK aufeinanderfolgen. Dabei wird auf die einzelnen Vektoren nicht noch einmal explizit eingegangen. Im Anhang sind alle Vektoren ausführlich dokumentiert.

DER KEY-MANAGER

Die Eingaben über die Tastatur entgegenzunehmen, ist Aufgabe des Betriebssystems, und zwar des Key-Managers. Aber ganz alleine kann auch er diese Aufgabe nicht bewältigen. Um die Tastatur zunächst überhaupt einmal "abzufragen" wird bei der Initialisierung des Rechners in die *Ticker Chain* ein Event-Block eingehängt, wodurch 50mal in jeder Sekunde die Tastatur überprüft wird. Der Key-Manager greift hier also auf den Software-Interrupt-Mechanismus des *Kernel* zurück.

Tastaturabfrage

Alle Tasten und auch die beiden Joystick-Anschlüsse sind an eine Drahtmatrix mit 8 mal 10 Leitungen angeschlossen. Normalerweise sind alle Taster geöffnet, und es besteht keine elektrische Verbindung zwischen den Zeilen- und den Spaltendrähten.

Die 10 Zeilendrähte sind mit den Ausgängen eines ICs 74LS145 verbunden. Das ist ein BCD-zu-Dezimal-Decoder mit offenen Kollektorausgängen. Normalerweise sind alle Ausgänge praktisch hochohmig. Über 4 Eingänge kann, binär codiert, aber ein Ausgang angewählt werden, der dann auf 0 Volt gezogen wird. Diese Eingänge sind an die Bits 0 bis 3 des Port C der PIO angeschlossen; die Ausgänge, wie gesagt, an die Zeilendrähte der Tastatur.

Demgegenüber sind die 8 Spaltendrähte an den I/O-Port des Sound-ICs angeschlossen. Intern sind diese Leitungen über einen Widerstand auf +5 Volt, also logischen Eins-Pegel hochgezogen. Der PSG wiederum ist an den Port A der PIO angeschlossen und muß noch über die Bits 6 und 7 des Port C angesteuert werden (BC1 und BDIR).

Um die Tastatur abzufragen, werden nun nacheinander alle Zeilendrähte via Port C und 74LS145 auf 0 Volt gelegt. Dabei wird jeweils mit denselben komplizierten Operationen wie bei der Programmierung von Geräuschen, über den Umweg über den I/O-Port des PSG (Sound-Chip) und über Port A der PIO, ein Byte eingelesen, das mit seinen 8 Bits den Zustand der Spaltendrähte repräsentiert.

Auf diese Weise werden für eine komplette Tastaturabfrage 10 Bytes (für 10 Zeilendrähte) eingelesen.

Normalerweise, wenn keine Taste gedrückt ist, werden alle Spaltendrähte bei jeder Abfrage auf logischem Eins-Pegel liegen; in den eingelesenen Bytes sind deshalb alle Bits gesetzt. Man erhält den Wert 255 = &FF = &X11111111.

Wird aber eine Taste gedrückt, so werden dadurch ein Zeilen- und ein Spaltendraht verbunden. Wird jetzt die Tastatur abgefragt, so wird auch weiterhin bei allen anderen Zeilendrähten das Byte &FF eingelesen, nicht aber dann, wenn der Zeilendraht aktiviert wird, der mit der gedrückten Taste verbunden ist. Dann wird nämlich auch der durch die Taste verbundene Spaltendraht auf 0 Volt gezogen, und an dieser Stelle enthält das eingelesene Byte eine Null.

Nach einer kompletten Tastaturabfrage erhält man also zunächst einmal nur 10 Bytes, in denen vielleicht das eine oder andere Bit null ist. Aus der Lage des Null-Bits läßt sich auf die gedrückte Taste zurückschließen.

Aus der Lage der einzelnen Tasten in der Drahtmatrix ergibt sich dann auch ihre Tastennummer, die man beispielsweise bei der BASIC-Funktion INKEY(nr) angeben muß. Die Tastennummern des ersten Bytes gehen von 0 bis 7, die des zweiten Bytes von 8 bis 15 usw. bis zum 10ten Byte. Die folgende Grafik zeigt die Lage der einzelnen Tasten in der Matrix und ihre Tastennummern. Dabei werden mit den Klammern folgende Lagen symbolisiert:

(...) → Zehnerblock oder Cursor-Taste
[...] → Joystick 0 (normaler Joystick)
{...} → Joystick 1 (zweiter Joystick)

Byte\Bit	0	1	2	3	4	5	6	7
0	(hoch)	(rechts)	(runter)	(9)	(6)	(3)	(ENTER)	(.)
8	(links)	(COPY)	(7)	(8)	(5)	(1)	(2)	(0)
16	CLR	[	ENTER	]	(4)	SHIFT	\	CTRL
24	^	–	@	P	;	:	/	.
32	0	9	O	I	L	K	M	,
40	8	7	U	Y	H	J	N	SPACE
48	6	5	R	T	G	F	B	V
! 48	{hoch}	{runter}	{links}	{rechts}	{Feuer1}	{Feuer2}	{n.c.}	
56	4	3	E	W	S	D	C	X
64	1	2	ESC	Q	TAB	A	CAPSLOCK	Z
72	[hoch]	[runter]	[links]	[rechts]	[Feuer1]	[Feuer2]	[n.c.]	DEL

Während der normale Joystick 0 einen eigenen Zeilendraht als COMMON-Leitung hat, ist COMMON 2 für den zweiten Joystick identisch mit einem Zeilendraht der Tastatur. Bei der Tastaturabfrage kann man deshalb nicht erkennen, ob eine Taste des zweiten Joysticks oder eine Taste der Tastatur gedrückt wurde.

Außerdem ist der Spaltendraht 6 am Joystick-Port herausgeführt. Er wird jedoch von keinem normalen Joystick benutzt und ist deshalb als n.c. (not connected – nicht angeschlossen) markiert. Wirklich nicht angeschlossen ist aber

die Taste mit dem Code 79. Hier könnte man so etwas wie eine Geheimtaste erfinden, um in die eigenen kopiergeschützten Programme wieder "hinein" zu können.

Entprellen

Es genügt jedoch nicht, die gedrückten Tasten zu erkennen. Mechanische Taster haben nämlich die unangenehme Eigenschaft, zu prellen. Wird die Taste gedrückt, so schließt der Kontakt, federt zurück, schließt und vielleicht dann noch einmal, bevor der Kontakt endlich fest hergestellt ist.

Beim Einlesen der Taste kann es also passieren, daß zunächst ein geschlossener Kontakt, dann wieder ein offener und dann wieder ein geschlossener erkannt wird.

Dieses Kontaktprellen muß nun Software-mäßig von bewußt wiederholten Tastendrücken des Anwenders unterschieden werden. Dazu dient die Zeit als Maß: Kontaktprellen ist naturgemäß nur eine sehr kurze Angelegenheit; Doppelanschläge auf der Tastatur dauern länger.

Die Tastatur des Schneider CPC wird dadurch entprellt, daß eine Taste erst dann als wieder geöffnet markiert wird, wenn sie bei zwei aufeinanderfolgenden Abfragen nicht mehr geschlossen war. Dadurch wird recht sicher erreicht, daß nur wirklich wieder losgelassene Tasten als wieder geöffnet markiert werden. Demgegenüber wird eine Taste aber sofort als gedrückt behandelt, wenn das entsprechende Bit null ist. Hier gibt es keine Verzögerung.

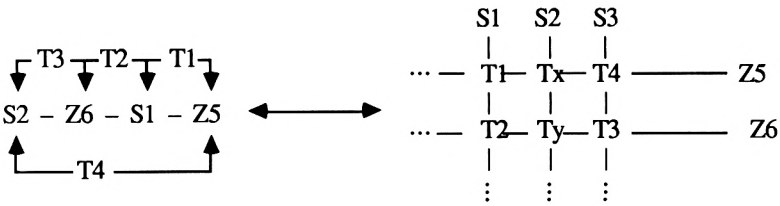
Drei im Karree

Eine weitere Eigenheit einer Matrix-förmig aufgebauten Tastatur ist zu beachten. Werden drei Tasten gleichzeitig gedrückt, die die Eckpunkte eines Rechtecks in der Tastenmatrix bilden, so wird der vierte, nicht gedrückte Eckpunkt auch ein Null-Bit liefern.

Als Beispiel sollen die Spaltendrähte 1 und 3 und die Zeilendrähte 5 und 6 betrachtet werden.

Durch Taste Eins wird S1 mit Z5 verbunden. Die zweite Taste verbindet S1 mit Z6, die dritte verbindet S3 mit Z6. Dadurch ist auch S3 mit Z5, die 4. Tastenposition, elektrisch verbunden, ohne daß die Taste gedrückt zu sein braucht.

Das müßte ebenfalls von der Tastatur-Software erkannt und abgefangen werden.



Dazu muß man einfach nur testen, ob eben vier oder noch mehr Tasten erkannt werden, und dann diese Abfrage einfach ignorieren. Da bei normalem Gebrauch der Tastatur maximal zwei Tasten gleichzeitig gedrückt werden, muß wohl bei drei oder noch mehr eingelesenen Tasten ein "Ausrutscher" beim Schreiben vorliegen.

In diesem Punkt ist die Tastatur-Software aber nicht sehr durchdacht gestaltet worden. Beim Schneider CPC wird nämlich genau dieser Fall nicht erkannt.

Als Beispiel kann man u.a. einen Fall anführen, der beim Erstellen von Assemblertexten manchmal auftreten kann:

Ist die Tastatur auf Kleinbuchstaben eingestellt und soll das Wort DEFW eingeben werden, so erscheint ziemlich oft:

```
D
EFW
```

Der Grund dafür ist, daß SHIFT, D und E mit ENTER ein Rechteck in der Tastaturmatrix bilden. Um DEFW einzugeben, hält man mit dem Daumen die SHIFT-Taste gedrückt. Ist man bei der Eingabe dann aber so schnell, daß E schon gedrückt wird, bevor D richtig losgelassen ist, dann sind alle drei Tasten gedrückt, und die vierte im Rechteck, nämlich ENTER, wird von selbst erzeugt.

Warteschlange

Die Tastaturabfrage auf dem Interrupt und das Programm, das die Eingaben bearbeitet, sind zwei unabhängig voneinander arbeitende Programme.

Damit ein Tastendruck seinen Weg zum laufenden Programm findet, sind zwei Arbeitsgänge notwendig: Zum einen muß der Key-Manager regelmäßig nach-

schauen, ob eine Taste gedrückt ist. Zum anderen müssen die erkannten Tasten vom laufenden Programm abgeholt werden. Es kann dabei durchaus vorkommen, daß das Programm eine Zeitlang keine Eingaben abholen kann, weil es gerade mit einer anderen Aufgabe beschäftigt ist.

In diesem Fall wird immer eine Warteschlange benötigt. An einem Ende werden bei der Tastaturabfrage Tasten nachgeschoben, am anderen Ende holt sich das laufende Programm die Tasten ab. Diese Queue ist dabei lang genug, 20 Tasten aufzunehmen. Programmtechnisch gesehen ist die Queue als Ringspeicher in einem Array für "Fixed Length Records" realisiert.

Ist der Puffer voll, so werden weitere Tastendrücke einfach ignoriert. Leider gibt der Key-Manager dann keine Warnung aus.

Tastenübersetzung

Natürlich ist es noch recht unkomfortabel, nur zu erfahren, welche Taste gerade gedrückt wurde. Meist ist es viel interessanter, zu erfahren, welches Zeichen dadurch erzeugt werden sollte.

Zu diesem Zweck hält der Key-Manager drei Tabellen bereit, in denen zu jeder Tastenposition eingetragen ist, welches Zeichen diese Taste erzeugen soll, wenn sie allein, zusammen mit SHIFT oder zusammen mit CTRL gedrückt wird. Zusätzlich gibt es noch eine Tabelle, in der eingetragen ist, welche Tasten sich automatisch wiederholen dürfen, wenn man seinen Finger lange genug daraufhält.

Diese Tabellen werden in das RAM kopiert und sind auch von BASIC aus änderbar. Dafür dient der Befehl

```
KEY DEF nr, rep, solo, shift, ctrl
```

Als Parameter müssen in dieser Reihenfolge die Tastennummer, der Repeat-Status (0 – keine automatische Wiederholung) und die Tastenübersetzungen allein, mit SHIFT und mit CTRL angegeben werden.

Repeat

Auch die Repeat-Funktion für die Tasten, denen es durch den Eintrag in der Repeat-Tabelle erlaubt wurde, sich zu wiederholen, ist sehr durchdacht implementiert worden: Zunächst wird zwischen der ersten Startverzögerung und der Wiederholungsverzögerung zwischen den folgenden Selbstanschlägen unterschieden.

Diese beiden Zeiten sind in BASIC mit "SPEED KEY startzeit, wiederholzeit" programmierbar. Eine Taste wird aber nicht bedingungslos neu angeschlagen, wenn die entsprechende Wartezeit abgelaufen ist. Der Key-Manager schaut erst nach, ob er auch keine weitere Taste mehr im Puffer hat. Nur wenn der Tastenpuffer vollständig leer ist, wird ein Repeat-Anschlag im Tastaturpuffer vermerkt. Wenn der Puffer nicht leer ist, wird die Taste nicht eingetragen und muß noch etwas warten.

Das ist sehr günstig, weil sich der Tastenpuffer so nicht unbemerkt mit selbstwiederholten Zeichen füllen kann, wenn das abarbeitende Programm einmal nicht schnell genug ist. Das wäre z.B. sehr ungünstig, wenn das mit der Delete-Taste geschieht. Hier hätte man dann sehr schnell sehr viel mehr Text gelöscht, als man eigentlich beabsichtigte.

CLEAR INPUT

Trotzdem kann sich im Tastaturpuffer aber auch "Müll" ansammeln, dann nämlich, wenn das Programm längere Zeit keine Eingaben entgegennehmen kann. Diese sammeln sich beim Key-Manager an.

Wendet sich das Programm dann wieder dem Anwender zu, so erhält es zunächst einmal den gesamten Müll aus dem Tastaturpuffer. Da der Anwender zu diesem Zeitpunkt diese "Eingaben" wahrscheinlich nicht mehr machen wollte, ist es sinnvoll, den Tastaturpuffer vorher zu leeren. Beim CPC 664 und 6128 gibt es dafür in BASIC den Befehl CLEAR INPUT. Aber auch beim CPC 464 kann man den Effekt mit kaum mehr Aufwand simulieren:

```
1000 WHILE INKEY$<>"":WEND
```

In Maschinensprache sieht das ähnlich einfach aus:

```
; Flush keyboard buffer
;
FLUSH:  CALL #BB09        ; KM READ CHAR (Hole Zeichen, falls vorhanden)
        JR   C,FLUSH      ; CY=1, wenn ein Zeichen da war, dann noch mal,
        RET               ; sonst fertig
```

Spezielle Tastencodes

In den Tasten-Übersetzungstabellen hat der Eintrag &FF eine spezielle Bedeutung: Hiermit wird angedeutet, daß diese Tastenkombination kein Zeichen erzeugen soll.

Aber es gibt noch weitere reservierte Zeichencodes: So wird mit &FE der Shift-Lock-Status und mit &FD der Caps-Lock-Status invertiert. Das sind die Funktionen, die man mit CAPS LOCK und CAPS LOCK plus CTRL erreicht.

&FC wird normalerweise von der ESC-Taste erzeugt. Zusätzlich wird aber auch das Zeichen &EF in den Tastenpuffer eingefügt, wenn der Break-Mechanismus aktiv ist. Das ist dafür gedacht, den Tastenpuffer bis zu der Stelle leeren zu können, an der der Anwender das Programm unterbrochen hat. In BASIC wird das aber nicht vom Anwenderprogramm, sondern vom BASIC-Interpreter vorgenommen.

Speziell die Behandlung der Break-Taste scheint beim CPC 464 noch nicht ganz ausgegoren zu sein. Beim CPC 664 und 6128 wurden hier einige Änderungen vorgenommen: So ist es beim CPC 464 noch möglich, die Zeichen &FC und &EF mittels INKEY$ einzulesen, wenn man eine Taste mit dieser Belegung definiert. Wenn man den Break-Mechanismus abschaltet (CALL &BB48) kann das sogar die ESC-Taste sein. Der CPC 6128 weigert sich aber, sowohl das Zeichen &FC als auch &EF auszugeben.

Beim CPC 464 enthält der BASIC-Interpreter noch einen Fehler, der es trotz ON BREAK GOSUB möglich macht, das Programm mit ESC zu stoppen. "Breaks", während der Zeileneditor auf eine Eingabe wartet (INPUT, LINE INPUT), stoppen in allen Fällen das Programm.

Das folgende Programm zeigt, wie man den Break-Mechanismus ganz ausschaltet, aber trotzdem noch Software-mäßig einen Break erkennen kann:

```
100 CALL &BB48                      ' entspricht ON BREAK CONT beim CPC
664/6128
110 '
120 KEY DEF 66,0,159,159,159    ' Break-Taste
130 KEY 159,CHR$(13)+CHR$(27)   ' erzeugt jetzt erst ENTER dann ESCAPE
140 '
150 ' Demo:
160 '
170 INPUT a                             ' laesst sich nicht breaken
180 IF INKEY$=CHR$(27) THEN GOTO 170    ' Test, ob man breaken wollte
190 FOR i=0 TO 1000:PRINT"#";:NEXT      ' laesst sich auch nicht breaken
```

Weitere Sonderbehandlungen nimmt der Key-Manager aber mit den Zeichen nicht mehr vor. Alle anderen Kontrollfunktionen, die einzelnen Zeichen bei der Arbeit mit dem Zeileneditor zugeordnet sind, werden nicht vom Key-Manager ausgewertet. Hier nimmt der Zeileneditor selbst die Zusatzinterpretationen vor.

Drei Abholmethoden

Neben der Möglichkeit, mit INKEY(nr) eine Taste direkt zu testen (wobei der Key-Manager nicht tatsächlich die Tastatur neu abfragt, sondern auf die 10 Bytes der letzten Abfrage zurückgreift), besteht also für das Vordergrund-Programm die Möglichkeit, sich schon fertig ausgewertete Zeichen von der Tastatur abzuholen.

Hierbei gibt es aber nochmals zwei unterschiedliche Möglichkeiten: Die eine, die unter BASIC normalerweise nicht zur Verfügung steht, ist die, daß man alle Zeichen so abholt, wie sie vom Anwender eingegeben wurden (außer der speziell interpretierten Zeichen &FC bis &FF). Hierfür muß man die Vektoren &BB18 und &BB1B benutzen.

Von BASIC wird jedoch ein anderer Vektor zum Key-Manager benutzt, der von der Möglichkeit Gebrauch gemacht, die Zeichen mit Codes von 128 (&80) bis 159 (&9F) expandieren zu lassen. Dabei werden diese Zeichen nicht weitergegeben, sondern in einer Übersetzungstabelle wird ein hierfür definierter String gesucht.

Hat der Benutzer durch einen Tastendruck ein solches "Erweiterungszeichen" erzeugt, so wird dem laufenden Programm mit jedem Aufruf des Vektors &BB06 oder &BB09 ein Zeichen aus dem entsprechenden Erweiterungs-String zurückgegeben, bis dieser vollständig abgearbeitet wurde.

Auch diese Tabelle ist im RAM angelegt und kann geändert werden. In BASIC geschieht das mit dem Befehl:

```
KEY nr,"string"
```

Zeichenrückgabe

Für Maschinensprache-Programme besteht darüber hinaus die Möglichkeit, ein Zeichen zurückzugeben. Man kann sich ein Zeichen vom Key-Manager abholen und dieses oder ein anderes zurückgeben.

In BASIC ist diese Möglichkeit nicht vorgesehen. Man kann aber die Erweiterungszeichen benutzen, um mit einigen Tricks hier doch etwas zu erreichen.

Wird nämlich ein Erweiterungszeichen gerade ausgegeben, so muß sich der Key-Manager ja selbst darüber auf dem Laufenden halten. Dazu benutzt er

zwei Speicherstellen seines System-RAMs, in die er die Nummer dieses Strings einträgt und welches Zeichen im String gerade dran ist.

	CPC 464	CPC 664/6128
Zeichen im $:	&B4DE	&B628
Exp$-Nummer:	&B4DF	&B629

Im folgenden Beispiel fügt ein BASIC-Programm sich selbst neue Zeilen ein:

```
 10 ' Selbst-veraenderndes BASIC-Programm     (c) G.Woigk vs.
    ' 22.5.86   (CPC 6128)
 20 ' ------------------------------
    '    ---------------------- ----------
 30 '
 90 PRINT
100 PRINT "Geben Sie die Formel ein:
110 PRINT
120 LINE INPUT "Y = ",a$
130 PRINT
140 INPUT "von X = ",x1
150 INPUT "bis X = ",x2
160 CLS
170 KEY 150,"500 y="+a$+CHR$(13)+"GOTO 220"+CHR$(13)
    ' Einfuegen der Zeile 500
180 PAPER 0:PEN 0
    ' und weiter in Zeile 220
190 POKE &B628,0              ' Key-Manager austricksen
200 POKE &B629,150
210 STOP                      ' hiernach liefert der KM die
                              ' Zeichen aus Zeile 170,
220 PEN 1:CLS                 ' und deswegen geht es hier
                              ' weiter
230 FOR x=x1 TO x2
500                           ' hier wird die neue Zeile
                              ' eingefuegt
510 PRINT "X =";x;"--> Y =";y
520 NEXT
600 RUN
```

DIE TEXT-VDU

Neben der Text-Eingabe ist wohl die Text-Ausgabe auf dem Bildschirm die zweite, wichtigste Schnittstelle zum Benutzer. Auch hierfür wurde im Schneider CPC eine eigene, fast autonome Abteilung des Betriebssystems geschaffen: Die Text-VDU.

VDU heißt dabei "Vector Driven Unit", also zeigergesteuerte Einheit, wobei hier mit Vektor nicht nur die Sprung-Vektoren im MAIN FIRMWARE JUMPBLOCK gemeint sind, sondern auch die Control-Codes von 0 bis 31.

Die Haupt-Einsprungstelle, über die man mit der Text-VDU kommunizieren kann, ist der Vektor &BB5A TXT OUTPUT. BASIC benutzt zum Bearbeiten der Print-Statements ausschließlich diesen Vektor.

Kontrollcodes

Von den insgesamt 256 verschiedenen Zeichen, die sich durch ein Byte voneinander unterscheiden lassen, werden die 32 ersten normalerweise nicht ausgedruckt, sondern als spezielle Steueranweisungen behandelt. Einfachstes Beispiel ist der Klingelton, der mit CHR$(7) erzeugt werden kann.

Damit ist es möglich, die wichtigsten Funktionen der Text-VDU mittels einfacher PRINT-Anweisungen auch von BASIC aus zu steuern. Unter anderem sind das Verschiebung des Cursors, Löschen von Bildschirm-Teilbereichen, Moduswechsel, Definieren von Zeichenmatrizen usw.

Darüber hinaus werden von der Text-VDU aber nicht nur diese leistungsfähigen Kontrollcode-Vektoren bereitgestellt. Sie lassen sich auch ändern! Zum Aufruf eines Kontrollcode-Handlers benutzt die Text-VDU nämlich eine Tabelle, die wieder einmal ins RAM kopiert wird.

Da die Kontrollcode-Behandlung durch ein Maschinencode-Programm erfolgen muß, ist diese Fähigkeit natürlich nicht bis zur BASIC-Ebene durchgeführt worden. Von Assembler aus stehen hier aber viele Möglichkeiten offen.

Dafür benötigt man zunächst einmal die Lage der Kontrollcode-Tabelle. Diese kann man mit dem Vektor &BBB1 TXT GET CONTROLS erfragen. Dann muß man wissen, wie die Tabelle aufgebaut ist.

Die Tabelle besteht aus insgesamt 32 Einträgen, für jeden Code einen. Der erste Eintrag ist der für den Code 0. Alle Einträge setzen sich wie folgt zusammen:

```
CTRLxx: DEFB ANZPAR     ; Anzahl Parameter
        DEFW ROUTADR    ; Adresse der Behandlungsroutine
```

Dabei muß die Behandlungsroutine bei eingeschaltetem unterem ROM erreichbar sein.

Die Zahl der Parameter darf bis zu neun Stück betragen. Beim CPC 664 und 6128 gibt es noch eine Besonderheit: Hier wird im Bit 7 der Parameterzahl ein Flag gespeichert, das angibt, ob die Ausführung des Kontrollcodes vom Enable/Disable-Status der Text-VDU abhängig sein soll (siehe CHR$(6) und CHR$(21)). Beim CPC 464 werden alle Kontrollcodes auch bei ausgeschalteter Text-VDU befolgt, beim CPC 664/6128 nur, wenn Bit 7 gesetzt ist. Dabei ist in der Standardbelegung die Ausführung fast aller Steuerzeichen vom VDU-Status abhängig (und damit nicht kompatibel zum CPC 464).

Die Behandlungsroutine eines Kontrollcodes ist als Unterprogramm der TXT OUTPUT-Routine aufzufassen. Sie wird von ihr aufgerufen, wenn der entsprechende Kontrollcode ausgedruckt werden sollte. Es gelten die folgenden Ein- und Aussprungbedingungen:

Ein – A und C = letzter Parameter
B = Anzahl der Parameter + 1 (für den Kontrollcode selbst)
HL zeigt auf den Kontrollcode-Puffer mit allen Parametern. Der erste Eintrag (auf den HL zeigt) ist der Kontrollcode selbst.
Aus – AF, BC, DE und HL dürfen verändert werden.

Beispiel: Piktogramm-Vektor

Das folgende Assemblerprogramm zeigt, wie man eine eigene Behandlungsroutine installieren kann. Gepatcht wird der Kontrollcode-Vektor CHR$(25), mit dem normalerweise Zeichenmatrizen neu definiert werden können.

Als neue Funktion wird eine Funktion installiert, mit der ein sogenanntes "Icon" ausgegeben werden kann. Dabei handelt es sich um kleine Piktogramme, mit denen man beispielsweise in einem Menü die einzelnen Optionen symbolisieren kann. Bekannt geworden sind diese "Icons" durch GEM, die grafikorientierte Benutzeroberfläche von Digital Research.

Und das soll der Kontrollcode CHR$(25) genau machen: Als Argument benötigt er einen Zeichencode z, der zusammen mit den folgenden drei Codes z+1 bis z+3 ein Pictogramm darstellt. Diese vier Zeichen werden auf der aktuellen Cursor-Position in Form eines Quadrates ausgegeben, und dann wird die Cursor-Position um zwei Stellen nach rechts verschoben.

Druckt man beispielsweise die folgenden beiden Zeichen aus:

```
PRINT CHR$(25);"A";
```

so sollen die Zeichen A bis D in Form eines Quadrates ausgedruckt werden:

```
AB
CD
```

Hätte man andere Zeichen benutzt und diese mit Hilfe des Befehls SYMBOL verändert, so könnte hiermit ein sinnfälliges Piktogramm, beispielsweise eine Diskette, Pinsel, Mülleimer etc. dargestellt werden.

Zum Verständnis des Programms: Die Ausgabe der Zeichen wird durch Aufruf der Indirection IND TXT OUT ACTION erreicht. Diese Indirection ist praktisch die Ausführungsroutine des Vektors TXT OUTPUT. Der Vektor besorgt ausschließlich das Einschalten des unteren ROMs (via Restart) und die Rettung der Register AF bis HL. Da beim Aufruf der Kontrollcode-Routinen das untere ROM immer eingeblendet ist, kann hier die Indirection benutzt werden, um Zeit zu sparen.

Die Kontrollcode-Routinen sind Unterprogramme von TXT OUTPUT (bzw. IND TXT OUT ACTION). Der Wiederaufruf der Indirection ist also praktisch eine Rekursion. Leider ist die Text-VDU nicht "reentrant". Sie darf nicht aufgerufen werden, solange sie noch eine Routine bearbeitet. Genau das wird aber benötigt.

Deswegen wird unsere Routine "transparent" eingebunden. Das heißt, sie ruft zunächst ihre eigene Rückkehradresse auf. Die Indirection meint, das sei ein ganz normaler Unterprogramm-Rücksprung (RET) gewesen und arbeitet nun ihren Programmcode fertig ab. Danach schließt auch die Indirection mit RET ab. Da sie von der Kontrollcode-Routine aufgerufen wurde, kehrt sie auch dahin zurück. Jetzt kann von hier aus die Indirection ohne Probleme aufgerufen werden, weil sie ja vollständig abgearbeitet ist.

```
; Patch von CHR$(25) --> Pictogramm-Vektor   vs. 22.5.86
; ---------------------------------------   -----------
;
        ORG  30000
;
; 0. Deklarationen:
;
ASKTRL: EQU  #BBB1          ; TXT GET CONTROLS
```

```
INDOUT: EQU  #BDD9          ; IND TXT OUT ACTION
JPHL:   EQU  #001E          ; LOW KL PCHL INSTRUCTION:   JP (HL)
;
; 1. Hier Relocalisator einbinden
;
; 2. Weitere Initialisierungen:
;
INIT:   CALL ASKTRL         ; Erfrage Lage der Kontrollcode-
;                           ; Tabelle → HL
        LD   DE,3*25        ; Adresse des Eintrags fuer CHR$(25)
        ADD  HL,DE          ; bestimmen
        EX   DE,HL          ; und nach DE (LDIR-Ziel)
        LD   HL,PATCH       ; HL = Zeiger auf Ersatz (LDIR-Quelle)
        LD   BC,3
        LDIR                ; und patchen
        RET
;
PATCH:  DEFB 1              ; 1 Argument
        DEFW CHAR25         ; Adresse der Routine
;
; 3. Kontrollcode-Behandlungsroutine
;
CHAR25: LD   (P1),A         ; A = 1. Argument = 1. Zeichencode des
;                           ; Piktogramms
        INC  A
        LD   (P2),A         ; ersten Zeichencode und die drei
;                           ; nachfolgenden
        INC  A              ; in den auszudruckenden Text einsetzen
        LD   (P3),A
        INC  A
        LD   (P4),A
;
        POP  HL             ; Rueckkehradresse vom Stack holen
        CALL JPHL           ; und aufrufen (transparent)
;
        LD   HL,TEXT        ; Text ab HL ausdrucken:
LOOP:   LD   A,(HL)         ; hole naechstes (erstes) Zeichen aus dem
;                           ; Text.
        AND  A              ; Test, ob Null als Schlussmarke
        RET  Z              ; A=0? dann fertig
        PUSH HL
        CALL INDOUT         ; sonst Zeichen drucken
        POP  HL
        INC  HL             ; Zeiger weiterstellen
        JR   LOOP           ; und mit naechstem Zeichen weitermachen
;
TEXT:
P1:     DEFB 0              ; Platz fuer Zeichencodes 1 und 2
P2:     DEFB 0              ; des Piktogramms
        DEFB 8,8,10         ; Cursor links, links und runter
P3:     DEFB 0              ; Platz fuer Zeichencodes 3 und 4
P4:     DEFB 0              ; des Piktogramms
        DEFB 11,0           ; Cursor hoch und Endemarke
```

Das folgende BASIC-Programm zeigt, wie man CHR$(25) jetzt einsetzen kann:

```
 98 ' Drei Beispiel-Piktogramme:
 99 '
100 DATA 00000000,00000000 , 00111111,11111100 ,
    ' 00000000,00000000
101 DATA 00000001,10000000 , 01111111,11111110 ,
    ' 01111110,01111110
102 DATA 00000010,01000000 , 11111111,11111111 ,
    ' 01111110,01111110
103 DATA 00001111,11110000 , 10000111,11111111 ,
    ' 00011110,01111110
104 DATA 00111111,11111100 , 10111111,11111111 ,
    ' 00011110,01111110
105 DATA 00000000,00000000 , 10001110,00111111 ,
    ' 01111111,11111110
106 DATA 00111111,11111100 , 10111101,11111111 ,
    ' 01111110,01111110
107 DATA 01110101,11101110 , 10000110,01110001 ,
    ' 01111100,00111110
109 DATA 10110101,11101101 , 11111111,10101111 ,
    ' 01111100,00111110
110 DATA 01110110,11101110 , 11111100,01101111 ,
    ' 01111110,01111110
111 DATA 00011010,11011000 , 11111111,11101111 ,
    ' 01111111,11110110
112 DATA 00011010,11011000 , 11000011,11110001 ,
    ' 01000000,01100110
113 DATA 00011010,10111000 , 11111000,01111111 ,
    ' 01011111,01110110
114 DATA 00001110,10110000 , 11111111,00001111 ,
    ' 01000000,01100010
115 DATA 00001111,11110000 , 01111111,11111110 ,
    ' 01111111,11111110
116 DATA 00000000,00000000 , 00111111,11111100 ,
    ' 00000000,00000000
120 '
130 RESTORE:s=256-12:SYMBOL AFTER s:a=HIMEM+1-8*s
    ' Vorarbeiten
135 '
140 FOR i=s TO s+2 STEP 2
150   FOR j=0 TO 7
160     FOR k=i TO i+8 STEP 4
    ' Zeichen definieren
165       FOR l=k TO k+1
    ' (etwas kompliziert,
170         READ b$:POKE a+l*8+j,VAL("&X"+b$)
    ' damit die DATA-Zeilen
175       NEXT
    ' einfach zu erstellen
180     NEXT
    ' waren)
190   NEXT
```

```
200 NEXT
210 '
220 MEMORY 29999:LOAD"CTRL25.BIN",30000:CALL 30000
    ' Maschinencode laden
230 MODE 1
    ' und initialisieren
240 LOCATE 30,2:PRINT CHR$(25);CHR$(s);
    ' Beispiel ausdrucken
250 PRINT "  ";CHR$(25);CHR$(s+4);
260 PRINT "  ";CHR$(25);CHR$(s+8);
```

Zeichensatz

Neben den 32 Kontrollcodes lassen sich natürlich auch normale Zeichen ausdrucken, und zwar 256 Stück. Oft wird behauptet, der Zeichensatz des Schneider CPC fange erst bei CHR$(32) (Space) an, weil darunter "nur" Kontrollcodes lägen. Das ist jedoch nicht wahr. Es ist nur ein wenig umständlicher, auch diese Zeichen auf den Bildschirm zu bekommen. In Maschinensprache ist das vergleichsweise einfach, weil man hier nur eine andere Einsprungstelle wählen muß: &BB5D TXT WR CHAR.

BASIC benutzt jedoch für seine PRINT-Befehle ausschließlich &BB5A TXT OUTPUT, und dieser Vektor befolgt die Steuerzeichen. Es ist aber trotzdem möglich, auch über diesen Vektor alle Zeichen auszugeben, weil dafür der Kontrollcode CHR$(1) vorgesehen ist:

```
100 FOR i=0 TO 255
110   PRINT CHR$(1);CHR$(i);"  ";
120 NEXT
```

Ein ähnliche Funktion hat das Steuerzeichen CHR$(5). Hiermit lassen sich auch alle Zeichen ausdrucken, nur daß sie hier auf der Position des Grafik-Cursors ausgegeben werden.

Die Text-VDU verfügt über einen Character-Generator für alle 256 Zeichen. Es ist aber auch möglich, von hinten her Zeichenmatrizen ins RAM zu kopieren, um sie dort verändern zu können. In BASIC dient dazu der Befehl SYMBOL AFTER.

Der BASIC-Interpreter reserviert den benötigten Platz über HIMEM. Da hier auch der Kassetten/Diskettenpuffer aufgemacht wird und auch Maschinencode-Programme vorzugsweise hier hinzugeladen werden, gibt es manchmal Probleme, die sich aber umgehen lassen (siehe Kapitel über die Speicheraufteilung im CPC).

Die Zeichenmatrix

Alle Zeichen, die so erzeugt werden können, sind in einer Matrix von 8 Zeilen zu 8 Spalten definiert. Diese werden im Speicher in 8 Bytes zu 8 Bits repräsentiert. Von den acht Bytes ist das erste jeweils für die oberste Rasterzeile zuständig, und innerhalb eines Bytes bestimmt das höchstwertige Bit die Punkte in der linken Spalte. Ein gesetztes Bit zeigt an, daß der entsprechende Punkt im Bildschirm in der Vordergrund-Farbe gesetzt werden muß, ein Null-Bit deutet Hintergrund an.

Wenn die Zeichen durch die Text-VDU auf den Bildschirm gemalt werden, so liegen die einzelnen Matrizen dicht an dicht neben- und übereinander. Der Zwischenraum zwischen den einzelnen Buchstaben auf dem Bildschirm muß also schon bei der Definition der Matrizen berücksichtigt werden.

Alle Zeichen des ROM-Zeichensatzes sind so aufgebaut, daß sie rechts und unten eine Pixelzeile Rand lassen, allerdings nicht bei Buchstaben mit Unterlängen, so daß beispielsweise zwischen einem kleinen g und einem Großbuchstaben darunter kein trennender Hintergrund mehr ist. Im Anhang ist der gesamte Standard-Zeichensatz dargestellt.

Fenster

Eins der besten Features der Text-VDU ist ihre Fähigkeit, bis zu 8 Textfenster (fast) unabhängig voneinander zu verwalten. Jedes Textfenster kann dabei einen beliebigen Ausschnitt des Bildschirms darstellen und funktioniert fast wie ein eigener kleiner Bildschirm.

"Fast" unabhängig sind diese einzelnen "Streams" deshalb, weil doch einige Seiteneffekte nicht ausgeschlossen wurden. So geht die Fenstertechnik noch nicht so weit, daß sich einzelne Fenster störungsfrei überschneiden können. Überlappen sich zwei Fenster, so wird der Schnittbereich einfach von Textausgaben etc. von beiden Fenstern beeinflußt.

"Fast" unabhängig voneinander auch deshalb, weil es nur einen gemeinsamen Kontrollcode-Puffer gibt. Steuerzeichen können ja bis zu neun Parameter beanspruchen. Wird mitten in einer PRINT-Sequenz das Textfenster gewechselt, so sind die Effekte nicht mehr ganz vorhersehbar. Unabhängig voneinander können aber eine ganze Reihe von Parametern eingestellt werden:

- Fenstergrenzen links/rechts/oben/unten
- Cursor-Position x/y

- Paper- und Pen-Tinte
- Cursor-Status: on/off – enabled/disabled
- VDU-Status: enabled/disabled
- Ausgabe auf der Grafikposition
- Hintergrund-Modus opaque/transparent

Innerhalb eines Textfensters kann der Cursor (des entsprechenden Fensters) vollkommen frei bewegt werden. Diese Fähigkeit kann man sich zunutze machen und in einer Textverarbeitung nicht nur hoch- und runter-scrollen, sondern auch Zeilen einfügen und löschen.

Force legal: Error

Leider hat auch hier das Betriebssystem einen kleinen Fehler.

Wenn der Cursor in der äußersten rechten Spalte eines Fensters steht und man noch ein Zeichen druckt, ohne einen Zeilenvorschub danach zu erzeugen, so steht jetzt der Cursor rechts außen von der Zeile. Ist der Cursor-Fleck ausgeschaltet, so wird die Cursor-Position noch nicht in das Textfenster zurückgezwungen.

Das wird erst gemacht, bevor das nächste Zeichen gedruckt wird. Dazu wird der Cursor in die erste Spalte der nächsten Zeile gestellt und dann erst das Zeichen gedruckt.

Was ist aber, wenn die nächsten Zeichen die Steuerzeichen CHR$(10) und CHR$(13) sind, mit denen der Cursor normalerweise in die erste Spalte der nächsten Zeile gestellt wird? Diese beiden Codes werden vom BASIC-Interpreter nach jeder PRINT-Anweisung erzeugt, die nicht mit einem Komma oder Semikolon abgeschlossen wird!

In diesem Fall dürfte der Cursor *nicht* vor Ausführung der Funktion in das Textfenster zurückgezwungen werden, weil sonst eine Leerzeile entsteht! Vor keinem einzigen Kontrollcode (außer solchen, die weitere Zeichen ausdrucken, wie CHR$(1)) darf der Cursor in die Textgrenzen zurückgezwungen werden, nur vor realen Textausgaben.

Kontrollcodes können total andere Funktionen haben, die z.B. SYMBOL-, PEN- oder PAPER-Statements ersetzen. Vor der Ausführung eines solchen Statements wird der Cursor ja auch nicht in das Textfenster zurückbefördert. Es kann auch passieren, daß man in der letzten Zeile außerhalb des Textfensters steht und nun per Steuerzeichen ein LOCATE ausführen will. Der Effekt ist: Das Fenster scrollt, bevor das LOCATE-Kommando befolgt wird.

Der Cursor

In BASIC ist es, zumindest beim CPC 464, nicht möglich, den Cursor anzuschalten. Der Grund ist, daß ihn der BASIC-Interpreter auf der Systemebene (on/off) ausschaltet. Da nutzt es nichts, daß man ihn auf der Anwenderebene (enabled/disabled) mit den Steuerzeichen CHR$(3) und CHR$(2) ein- und wieder ausschalten kann.

Der Cursor wird nur dargestellt, wenn beide Ebenen an sind. Durch Aufruf der Vektoren &BB81 TXT CURSOR ON und &BB84 TXT CURSOR OFF erlangt man aber auch von BASIC aus Kontrolle über den Cursor-Fleck. Diese Vektoren haben keine Ein- und Ausgabebedingungen und sind deshalb für den Aufruf von BASIC aus geeignet.

Betrachten Sie einmal das folgende Programm: INPUT ohne Cursor:

```
100 PRINT CHR$(2);
110 INPUT a
```

DIE GRAFIK-VDU

Die zweite große Firmware-Abteilung, die den Bildschirm als Ausgabemedium benutzt, ist die Grafik-VDU. Hier sind alle Routinen zusammengefaßt, mit deren Hilfe der Aufbau von Grafik auf dem Bildschirm erleichtert wird. Zunächst einmal gibt es auch für die Grafikausgabe ein Fenster, das nach Belieben in den Bildschirm gelegt werden kann. Auch bei der Ausgabe von Buchstaben auf der Position des Grafik-Cursors dient dieses Fenster als Begrenzung.

Als Funktionen stellt die Grafik-VDU beim CPC 464 hauptsächlich Befehle bereit, um Linien zu ziehen, Punkte zu setzen und zu testen. Die Koordinatenangaben sind dabei jeweils absolut (im Bezug auf einen Ursprung, den "ORIGIN") oder relativ zur letzten Zugposition zu machen. Diese "letzte Zugposition" wird in Grafik-System-RAMs gespeichert und als "Grafik-Cursor" bezeichnet.

Die Koordinaten

Bei den Koordinatenangaben sind auf dieser Ebene drei Möglichkeiten zu unterscheiden:

- absolut (zum Origin)
- relativ (zur letzten Position)
- und Standardkoordinaten (zur linken unteren Ecke des Bildschirms)

Gemeinsam ist bei allen Systemen, daß sie mit Integer-Werten von –32768 bis +32767 arbeiten. Der Bildschirm ist unabhängig vom momentan gewählten Modus immer 400 Koordinateneinheiten hoch und 640 Koordinateneinheiten breit.

Daraus ergibt sich, daß immer zwei Y-Koordinaten auf dieselbe Rasterzeile des Bildschirms zugreifen. Je nach Modus entsprechen in X-Richtung 1, 2 oder 4 Koordinateneinheiten demselben Pixel.

Standardkoordinaten werden benutzt, um das Grafikfenster zu definieren. Dabei kann man die linke und die rechte Fenstergrenze nicht beliebig fein wählen. Das geht nur in Achter-Schritten. Grund dafür ist der physikalische Bildschirmaufbau, der aber erst beim Screen Pack erläutert wird.

Dabei wird die linke Fenstergrenze immer modulo 8 herabgesetzt, also nach links ausgeweitet, und die rechte Fenstergrenze modulo 8 + 7 nach rechts ausgeweitet:

```
ORIGIN 0,0,13,563,399,0

links  =  13 - ( 13 mod 8)      = 8*( 13 \ 8)              = 8
rechts = 563 - (563 mod 8) + 7 = 8*(563 \ 8) + 7 = 560+7 = 567
```

Aber auch für die Ober- und Untergrenze gilt ähnliches, weil hier immer zwei Y-Koordinaten einer Rasterzeile auf dem Monitor entsprechen: Die Untergrenze wird zur nächsten geraden Zahl abgerundet (modulo 2), und die Obergrenze wird zur nächsten ungeraden Zahl aufgerundet (modulo 2 + 1).

Der Linien-Algorithmus

Wenn man den Draw-Befehl verwendet, braucht man eigentlich nicht zu wissen, wie der Computer die einzelnen Punkte einer Linie berechnet. Wir wollen dieser Frage trotzdem einmal nachgehen.

Eine beliebte Vermutung ist, der Computer bestimme die Entfernung zwischen den beiden Punkten, berechne daraus die Anzahl der benötigten Punkte, daraus die Schrittweite in X- und Y-Richtung und gehe dann in diesem Raster von einem Punkt auf den anderen zu:

```
100 MODE 1                      ' Modus 1 -> Pixelabstand dx und dy = 2
105 PRINT CHR$(23);CHR$(1);     ' XOR-Grafik-Vordergrund-Modus
110 xa=50:ya=200
120 xe=600:ye=30
130 GOSUB 160                   ' mit Beispielwerten aufrufen
140 GOTO 140
```

```
150 '
151 ' Der Algorithmus:
152 '
160 anzp=SQR(ABS((xa-xe))^2+ABS((ya-ye))^2)/2 ' Bestimmung der Diagonalen
170 '                                          ' von (xa,ya) nach (xe,ye)
180 dx=(xe-xa)/anzp          ' X-Schrittweite
190 dy=(ye-ya)/anzp          ' Y-Schrittweite
200 FOR n=1 TO anzp
210   xa=xa+dx:ya=ya+dy      ' xa und ya auf xe bzw. ye zubewegen
220   PLOT xa,ya             ' und Punkt setzen
230 NEXT
240 RETURN
```

Läßt man dieses Programm ohne Zeile 105 laufen, so ist das Ergebnis ganz akzeptabel. Mit Zeile 105 wird aber ein Fehler offenbar: Die Anzahl der zu setzenden Punkte ergibt sich nicht als die Länge der Hypothenuse im rechtwinkligen dx-dy-Dreieck. Hier werden zu viele Punkte geplottet; manche heben sich deshalb wieder auf (Zeile 105 stellt den XOR-Modus ein).

Die Methode hat aber noch zwei andere Nachteile, die jeder Maschinencode-Programmierer gerne umgeht: Erstens wird mit Fließkommazahlen gerechnet (mit Integer funktioniert es nicht mehr), und zweitens muß man auch dividieren. Das sollte man vermeiden.

Der im Schneider CPC verwendete Algorithmus kommt nur mit Addition und Subtraktion von Integer-Werten aus. Daß im nun folgenden Beispiel trotzdem Multiplikationen mit 2 auftauchen, liegt daran, daß dies der Pixelabstand in Modus 1 ist. Diese Multiplikation läßt sich aber auch einfach durch Shiften eines Registers erreichen!

```
160 dx=xe-xa:dy=ye-ya
170 if abs(dx)<abs(dy) then f=1 else f=2:gosub 450:gosub 190:goto 460
180 '
190 if dy<0 then gosub 400:gosub 200:goto 410
200 z=dy\2:x=xa:dz=abs(dx):dx=2*sgn(dx)
205 '
210 for y=ya to ye step 2
220   z=z+dz:if z>=dy then x=x+dx:z=z-dy
230   if f=1 then plot x,y else plot y,x
240 next
250 return
390 '
400 dx=-dx:dy=-dy     ' Vertausche Anfang und Ende, weil dy<0
410 z=ya:ya=ye:ye=z   ' dadurch ist ye immer groesser (ueber) ya
420 z=xa:xa=xe:xe=z   ' der Algorithmus arbeitet immer mit wachsenden y
430 return
440 '
450 z=dx:dx=dy:dy=z   ' Vertausche x und y, weil dx>dy
460 z=xa:xa=ya:ya=z   ' dadurch arbeitet der Algorithmus immer in Richtung
470 z=xe:xe=ye:ye=z   ' der Y-Achse und variiert in X-Richtung.
480 return            ' Das Vertauschen von x und y wird im Flag f ange-
                        merkt und beeinflusst das Plotten in Zeile 230
```

Dieses Programm entspricht zwar nicht ganz exakt dem Algorithmus des Schneider CPC, kommt ihm aber schon sehr nahe. Die gravierendsten Unterschiede ergeben sich daraus, daß der Linien-Algorithmus der Grafik-VDU natürlich stärker auf die Hardware-Gegebenheiten des Bildschirms zugeschnitten ist.

Der Grundgedanke ist, daß man in Richtung der größeren Differenz vorgeht. Ist dy also größer als dx, so geht man in Y-Richtung vor. In Y-Richtung muß man dy Schritte machen. Zwischendurch muß man aber auch mal in Querrichtung vorwärtsgehen, und zwar dx-mal.

Um zu berechnen, wie oft ein solcher dx-Schritt in dy-Richtung kommen muß, teilt man nur dy durch dx. Nun kann man dazu streng nach Vorschrift dividieren und muß dann aber auch die Nachkommastellen berücksichtigen. In diesem Algorithmus steckt die offensichtliche Beziehung (dx*dy)/dx = dy.

Zunächst einmal wird das Ganze von der Schleife über dy, die längere Strecke, umfaßt (Zeile 210 bis 230). Bei jedem Durchgang wird in Zeile 220 zu der Variablen z einmal dz addiert. dz ist aber abs (dx) (siehe Zeile 200). Vom ersten bis zum letzten Durchgang wird also dy-mal der Wert dx addiert, also dx*dy!

Jedesmal wenn nun z den Betrag von dy erreicht oder übersteigt, wird wieder dy abgezogen. Dieser Fall wird genau (dx*dy)/dy, also dx-mal im Verlauf des Linien-Plottens auftreten. (Wie man sieht: Hier wird nicht nur multipliziert, indem man beständig addiert, sondern auch dividiert, in dem man ständig subtrahiert.) Immer wenn z den Wert von dy überschreitet, wird deshalb einmal zur Seite gesteppt: x=x+dx, wobei dx in Zeile 200 als 2*sgn(dx) definiert wurde, das ist die Schrittrichtung, und wegen Bildschirmmodus 1 in der Schrittweite von 2.

Fehler 1 (korrigiert)

Es gibt allerdings noch weitere Unterschiede zwischen dem CPC-Algorithmus und dem gerade vorgestellten.

Zwei Fehler wurden beim CPC 664/6128 behoben. Der erste betrifft den ersten Punkt einer Linie. Mathematisch korrekt ist es, den ersten Punkt einer Linie nicht zu zeichnen! Nehmen wir einmal an, es soll ein Kreis gezeichnet werden. Dieser wird beispielsweise durch 50 einzelne Linien approximiert. Es müssen also 50 Linien von P1 nach P2 nach P3 ... nach P50 und wieder nach P1 gezeichnet werden.

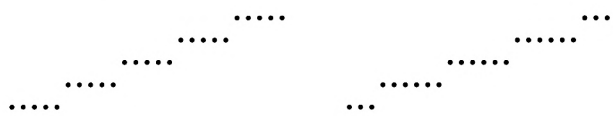
Der Endpunkt einer Linie ist der Startpunkt der nächsten; diese "Eckpunkte" werden also doppelt gezeichnet, wenn man sowohl den ersten als auch den letzten Punkt jeder Linie setzt!

Das ist meist nicht so schlimm. Wenn man aber mit dem Vordergrund-Modus XOR arbeitet, werden diese Eckpunkte zu Löchern in der Kreislinie. Und dabei ist der XOR-Vordergrund-Modus durchaus brauchbar: Hiermit läßt sich nämlich "löschbar" zeichnen: Zum Löschen eines Linienzugs muß dieser im XOR-Modus einfach ein zweites Mal ausgeführt werden. Die Löcher im Kreis fallen vielleicht zunächst kaum auf. Will man den Kreis aber nachher mit einem Fill-Algorithmus ausmalen, so "läuft" dieser durch die Löcher "aus".

Fehler 2 (korrigiert)

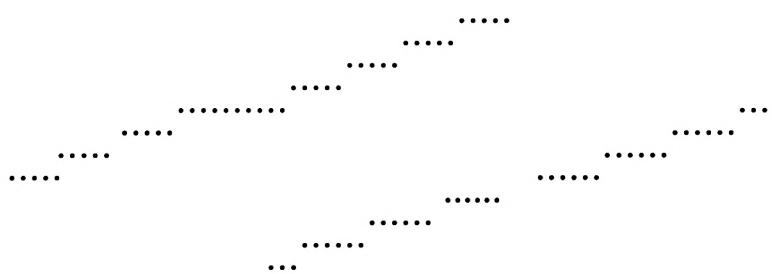
Zweiter Unterschied des CPC-Linien-Algorithmus beim CPC 464 zum hier vorgestellten ist: Der CPC 464 unterteilt die größere Differenz (dx oder dy) in dy+1 (dx+1) Schritte! Dadurch werden die einzelnen Linien etwas gefälliger. Mathematisch gesehen ist das nicht korrekt. Die folgende Grafik verdeutlicht den Unterschied:

CPC-464-Methode: korrekt:



Die kürzere Differenz dy beträgt in diesem Beispiel dy=4 Schritte. Der CPC unterteilt die Linie in 5 Stufen, korrekt wäre 4 = 1/2 + 3 + 1/2. Daß die CPC-464-Methode falsch ist, erkennt man, wenn man zwei dieser Linien zusammensetzt:

zweimal CPC-464-Methode: zweimal korrekt:



Wird bei der dx+1-Methode ein Linienzug zweigeteilt, so ergibt sich an der Schnittstelle ein überlanger Abschnitt. Das kann man auch auf dem CPC 464 mit folgendem Programm zeigen:

```
100 PLOT 0,100
110 DRAWR 300,10
120 DRAWR 300,10
130 GOTO 130
```

Fehler 3

Bei den Koordinaten gibt es beim Null-Durchgang sowohl in X- als auch in Y-Richtung einen Stetigkeitssprung!

Alle Koordinaten, die man bei PLOT, MOVE und TEST angibt, werden zunächst einmal nach Integer gewandelt. Dadurch wird beispielsweise die Koordinate +67.5 nach +68, –55.5 nach –56 (und nicht –55) und +34.4 nach +34 gerundet.

Die Zwangsrundung von Fließkommazahlen nach Integer geschieht dabei mit der Funktion ROUND(x).

Bereits die Tatsache, daß Zahlen mit einer 5 hinter dem Dezimalpunkt im positiven Bereich auf- und im negativen Bereich abgerundet werden, bringt beim Null-Durchgang eine Unstetigkeit mit sich.

Viel schlimmer macht sich aber folgendes bemerkbar:

Da die realen Bildschirmpunkte (Pixels) in Y-Richtung immer zwei und in X-Richtung je nach Modus 1, 2 oder 4 Koordinaten umfassen, muß auch hier festgelegt werden, welches Pixel nun mit PLOT 101, –3 eingefärbt werden soll. Dabei ist man bei AMSTRAD darauf verfallen, immer zur Null hinzurunden. Was das bringt, zeigt das folgende Programm:

```
100 MODE 1                    ; → Pixel-breit in X-Richtung: 2
                              ; Koordinaten
110 PRINT CHR$(23);CHR$(1);   ; XOR-Modus
120 ORIGIN 320,200            ; Origin ((0,0)-Kreuz) in die
                              ; Bildschirmmitte
130 FOR x=-50 TO +50 STEP 2
140   PLOT x,10               ; Ergebnis o.k.
150 NEXT
160 '
170 FOR x=-49 TO +49 STEP 2
```

```
180   PLOT x,0                ; Luecke beim Null-Durchgang!
190 NEXT
200 GOTO 200
```

In der ersten Schleife mit x von -50 bis +50 läuft alles glatt. Bei der zweiten Schleife mit ungeraden Werten gibt es plötzlich eine Lücke beim Null-Durchgang.

Der Grund dafür ist, daß in Modus 1 normalerweise zwei X-Koordinaten dasselbe Pixel ansprechen, nur nicht beim Null-Durchgang: Hier werden sowohl +1 als auch –1 zur 0 hin gerundet:

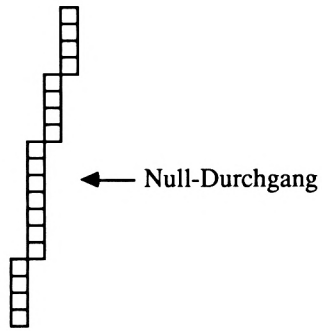
```
  –5   und –4
  –3   und –2
–1,  0 und +1
+2     und +3
+4     und +5
```

Dadurch wird beim Null-Durchgang mit den Werten X= –1 und X= +1 dasselbe Pixel angesprochen, und im XOR-Modus hebt sich das wieder auf.

Aber auch ohne XOR-Modus bringt das Fehler:

```
120 MODE 0
110 ORIGIN 320,200
120 x=-25
130 FOR y=-50 TO 50 STEP 2
140   PLOT x,y:x=x+1
150 NEXT
160 GOTO 160
```

Das Ergebnis sieht etwa so aus:



Weil 4 Koordinateneinheiten in X-Richtung sich jeweils auf dasselbe Pixel beziehen, muß x viermal erhöht werden, bevor es in dieser Richtung zu einer Änderung kommt, nur beim Null-Durchgang nicht. Hier werden 7 X-Koordinaten zur Null hingerundet: von –3 bis +3.

Wenn dieser Fehler Probleme bereitet, muß man auf die ansonsten sehr nützliche Möglichkeit, den Grafik-Origin zu verlegen, verzichten und die Koordinatenachsen wieder aus dem Bild verbannen.

Die Linienmaske

Beim CPC 664 und 6128 wurde die Grafik-VDU nicht nur von Fehlern befreit, sondern auch noch um einige Features erweitert.

So kann man jetzt (neben der Erste-Punkt-Option) eine Punktmaske für gesetzte und nicht gesetzte Punkte in einer Linie bestimmen. Diese wird mit jedem neu zu setzenden Punkt einmal rotiert, und so wird das nächste Bit in ihr "angewählt". Das bestimmt dann, ob ein Punkt in der aktuellen Vordergrund-Tinte der Grafik-VDU mit dem Vordergrund-Modus gezeichnet werden soll (Bit = 1) oder mit Hintergrund-Farbe und im Hintergrund-Modus.

Zwischen zwei aufeinanderfolgenden DRAW-Befehlen wird die Maske nicht wieder in Null-Position rotiert, sondern so, wie sie steht, weiterverwendet. Dadurch gibt es dann keinen Bruch im Muster.

Leider macht die Grafik-VDU beim Zeichnen von Linien eine Vereinfachung, die beim Linien-Algorithmus angedeutet wurde: Alle eher waagerechten Linien werden von links nach rechts und alle eher senkrechten Linien von unten nach oben gezeichnet, auch wenn der Zielpunkt weiter links bzw. weiter unten liegt. Auch die Maske wird dann in dieser Richtung angewendet, so daß es doch zum Musterbruch kommen kann.

Hintergrund-Modus

Weiter kommt beim CPC 664 hinzu, daß man nun auch für die Grafik-VDU einen Hintergrund-Modus definieren kann. Wie bei der Text-VDU kann man dabei zwischen OPAQUE und TRANSPARENT wählen. Der Hintergrund-Modus wirkt auch auf die Buchstaben, die auf die Position des Grafik-Cursors gezeichnet werden, wodurch sich hier Sprites (Shapes) besser realisieren lassen.

Der Fill-Algorithmus

Eine sehr nützliche Funktion, die mit dem CPC 664 eingeführt wurde, ist der schnelle Füll-Algorithmus. Er arbeitet nicht rekursiv wie der bereits vorgestellte BASIC-Einzeiler, sondern er speichert nur "interessante Punkte".

Um das zu verstehen, muß man sich einmal Gedanken machen, wie so eine Ausmal-Routine prinzipiell funktionieren muß.

Die Routine bekommt eine Startkoordinate übergeben, die in einer beliebig umgrenzten Fläche liegt. Aber es gibt rundherum eine Grenze. Man muß also

testen und unterscheiden können, ob ein Punkt "gesetzt" oder "nicht gesetzt" ist. Im ersten Fall begrenzt er die Ausmal-Routine, im zweiten Fall muß er gesetzt werden.

Außerdem müssen auch die nur diagonal verbundenen Grenzpunkte als "dicht" angesehen werden. Der im CPC verwendete Linien-Algorithmus produziert Linien, die sehr oft nur diagonal verbundene Punkte enthalten. Daraus ergibt sich die Forderung, daß man sich beim Pixel-Testen nur in den vier Grundrichtungen, nicht aber diagonal bewegen darf.

Füllen mittels Iteration

Nehmen wir einmal an, die Routine würde ein Pixel auf einer bestimmten Position testen und feststellen, daß es noch nicht gesetzt ist. Zunächst einmal muß sie dann den Punkt einfärben. Dazu muß sie aber auch die vier Nachbarpunkte testen, ob diese auch nicht gesetzt sind. Da man diese nicht gleichzeitig untersuchen kann, muß man die Koordinaten dieser Punkte speichern.

Aber auch jetzt ist der Speicherbedarf enorm: Ein Punkt hat vier Nachbarn, die zusammen 16 haben (daß dadurch einige mehrfach abgedeckt werden, ist bei einem Algorithmus schwer zu berücksichtigen), die 64 potentielle Nachbarn haben dann 256, 1024, 4096 etc.

Was passiert, wenn der getestete Punkt gesetzt ist? Dann wird die Rekursion oder das explodierende Abspeichern von Nachkoordinaten einfach abgebrochen und nichts mehr getan.

Das folgende Programm erstellt in Zeile 100 bis 250 eine Demo-Grafik und ruft das Unterprogramm ab Zeile 260 auf, um die Fläche auszufüllen:

```
100 MODE 1:defint a-z
110 dx=638:dy=398:MOVE 0,0:GOSUB 240
120 MOVE 10,390:dx=10
130 FOR dy=-10 TO -380 STEP -12
140 GOSUB 240:MOVER 20,0
150 NEXT
160 r=100:ORIGIN 10,10
170 DEG:MOVE -50,150
180 FOR w=-20 TO 130 STEP 5
190 DRAW SIN(w)*r,COS(w)*r
200 r=400-r
210 NEXT
220 x=400:y=20:GOSUB 260
230 GOTO 230
240 DRAWR dx,0:DRAWR 0,dy:DRAWR -dx,0:DRAWR 0,-dy:RETURN
250 '
260 IF TEST(x,y) THEN RETURN          ' wird ueberhaupt etwas gefuellt?
```

```
270 d=200:DIM x(d),y(d):z1=0:z2=1      ' Ringspeicher definieren
280 x(0)=x:y(0)=y                      ' ersten Wert eintragen

290 IF z1=z2 THEN RETURN               ' Test auf Ende
300 x=x(z1):y=y(z1):z1=(z1+1)MOD d     ' Wert aus Speicher holen, Zeiger
                                       ' incr.
310 x=x-2       :GOSUB 400             ' nun alle
320 x=x+4       :GOSUB 400             ' Nachbarpunkte
330 x=x-2:y=y-2:GOSUB 400              ' testen
340       y=y+4:GOSUB 400
350 GOTO 290                           ' und beim nächsten Punkt
                                       ' weitermachen
360 '
400 IF TEST(x,y) THEN RETURN           ' Karteileiche oder Umrandung?
410 PLOT x,y                           ' sonst Punkt setzen
420 x(z2)=x:y(z2)=y                    ' und in Speicher eintragen
430 z=(z2+1)MOD d:IF z<>z1 THEN z2=z   ' Zeiger nur erhoehen, wenn
                                       ' Speicher nicht
440 RETURN                             ' voll
```

Zum Speichern der Nachbarkoordinaten wird ein Ringspeicher benutzt, der in Zeile 270 definiert und in Zeile 280 mit dem ersten (Start-) Wert gefüllt wird.

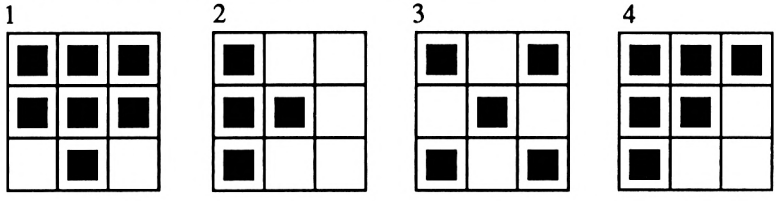
Normalerweise benutzt man einen Stack. Der hat aber den Nachteil, daß vielfach eingetragene und anderweitig bereits bearbeitete Punkte fast beliebig lange gespeichert bleiben können, bis der Stack ganz zum Schluß geleert wird.

Solchermaßen realisierte Paint-Routinen erkennt man daran, daß sie beim Ausmalen regelmäßig Pausen einlegen, weil der Speicher voll ist und von allen Karteileichen bereinigt werden muß.

Bei der Queue kommen aber regelmäßig alle Einträge dran. Dadurch werden ganz automatisch tote Punkte entfernt.

Interessante Punkte

Der CPC 664/6128-Füller arbeitet aber noch ein wenig anders: Er speichert nur "interessante" Punkte. Was unterscheidet einen "interessanten" von einem "unwichtigen" Punkt? Betrachten Sie dazu die folgenden Bilder. In den 3*3 - Feldern ist jeweils der mittlere Punkt gesetzt worden:



Der Algorithmus soll so funktionieren, daß nicht der gerade gesetzte Punkt gespeichert wird, sondern die vier umliegenden noch zu testenden Punkte.

In Bild 1 müssen keine Punkte in den Speicher aufgenommen werden, weil alle Nachbarn bereits gesetzt sind.

In Bild 2 kommen die Punkte (o)ben, (r)echts und (u)nten in Frage. Aber nur einer muß in die Tabelle eingetragen werden, da diese Punkte über die diagonalen Eckpunkte verbunden sind. Wird mit (o) weitergemacht, so findet man über dessen rechten Nachbar und dann dessen unterem Nachbar auch zu (r). Entsprechendes gilt für (u).

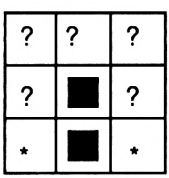
Unter diesem Gesichtspunkt betrachtet, müssen in Bild 3 aber alle vier Nachbarpunkte gespeichert werden, weil alle Eckpunkte bereits gesetzt sind und diese die Nachbarn voneinander trennen.

In Beispiel 4 muß wieder nur ein Punkt weiter betrachtet werden, weil (r) und (u) über den Eckpunkt verbunden sind.

Füllen mit Iteration und interessanten Punkten

Das folgende Programm macht dabei noch zwei weitere Vereinfachungen: Zum einen werden nur Verzweigungen in die Tabelle aufgenommen. Also nur, wenn zwei Pfade weiter verfolgt werden müssen, wird der eine zwischengespeichert. Mit dem anderen wird direkt weitergemacht.

Zum anderen wird die aktuelle Zugrichtung festgehalten. Dabei dreht der Algorithmus wenn möglich nach jedem Punkt einmal nach links. Dadurch müssen nicht mehr alle Nachbarpunkte untersucht werden.



In diesem Beispiel zog der Algorithmus von (u) in die Mitte (und drehte die Zugrichtung nach links). Wenn man jetzt von der Mitte aus die Nachbarpunkte untersucht, weiß man sicher, daß (u) gesetzt ist. Dadurch müssen (ul), (u) und (ur) nicht mehr untersucht werden.

```
260 DIM x(200),y(200),dx(200),dy(200)
270 dx=2:dy=0:zg=0
280 IF TEST(x,y) THEN 450
290 PLOT x,y:z=dy:dy=dx:dx=-z      ' Plot und drehe nach links

300 IF TEST(x+dx,y+dy)=0 THEN 390 ' Punkt frei? dann nach links
                                   ' weitermachen
```

```
310 z=dx:dx=dy:dy=-z                ' sonst wieder nach rechts drehen
320 IF TEST(x+dx,y+dy)=0 THEN 360   ' Punkt frei? dann geradeaus
                                    ' weitermachen.
330 z=dx:dx=dy:dy=-z                ' sonst nochmal nach rechts drehen
340 x=x+dx:y=y+dy                   ' vorwärts gehen (insgesamt nach
                                    ' rechts)
350 GOTO 280                        ' und weiter; Verzweigungen nicht
                                    ' moeglich!

355 ' Punkt rechts auf Verzweigung testen, nach vorne schreiten und
weiter:

360 IF TEST(x+dy,y-dx)=0            ' Test auf Verzweigung:
      THEN IF TESTR(dx,dy)          ' rechts frei aber rechts vorne
                                    ' gesetzt?
              THEN GOSUB 380        ' wenn ja, eintragen

370 x=x+dx:y=y+dy:GOTO 290          ' Schritt nach vorne und weiter

380 SOUND 1,200,20:                 ' Punkt rechts eintragen
    zg=zg+1:x(zg)=x+dy:y(zg)=y-dx:dx(zg)=dy:dy(zg)=-dx:
    RETURN

385 ' Punkt hinten auf Verzweigung testen und dann weiter bei 360:

390 IF TEST(x-dx,y-dy) THEN 360     ' Punkt hinten gesetzt? dann keine
                                    ' Verzweigung
400 IF TESTR(dy,-dx)=0              ' sonst: trennt Punkt rechts oder
    AND TESTR(dx,dy)=0 THEN 360     ' Punkt rechts hinten? nein, dann keine
                                    ' Verzw.
410 SOUND 1,200,20:                 ' sonst erst Punkt hinten eintragen
    zg=zg+1:x(zg)=x-dx:y(zg)=y-dy:dx(zg)=-dx:dy(zg)=-dy:
    GOTO 360

450 SOUND 1,300,20:                 ' Punkt aus Speicher holen:
    IF zg THEN x=x(zg):y=y(zg):dx=dx(zg):dy=dy(zg):zg=zg-1:GOTO 280
460 RETURN                          ' Zeiger zg=0, dann fertig
```

Zwar entspricht dieses Programm auch noch nicht dem CPC-Algorithmus, es zeigt aber ein Beispiel, wo nur noch Verzweigungspunkte in die Tabelle eingetragen werden. Die Füllroutine ist so gestaltet, daß man jedes PUSH und POP auf dem Verzweigungs-Stapel mitbekommt: PUSH – hoher Ton, POP – tiefer Ton.

Die Routine setzt einen Punkt, dreht sich nach links und testet, ob sie in dieser Richtung weitermachen kann. Wenn nicht, dreht sie wieder zurück nach rechts (und zeigt so wieder in die ursprüngliche Richtung), prüft diese Richtung und dreht sogar eventuell noch einmal nach rechts.

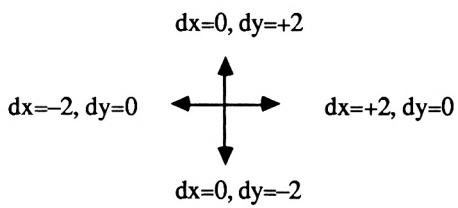
Im letzten Fall kann keine Verzweigung vorliegen. Von hinten kommt sie ja, links und vorne sind dicht (weshalb sie ja überhaupt nur bis rechts drehen mußte), allenfalls nach rechts kann es noch weitergehen.

Geht die Routine geradeaus, muß nur nach rechts auf eine Verzweigung getestet werden, denn nach links ist der Weg versperrt (sonst würde sie ja da weitermachen). Eine Verzweigung liegt vor, wenn es nach rechts weitergeht, die Verbindung nach vorne aber durch ein gesetztes Pixel rechts vorne unterbrochen ist.

Ist aber schon der Weg nach links frei, so müssen erst noch die Verzweigungen nach vorne und nach rechts überprüft werden.

Vektordrehung

Zum Verständnis der Routine ist noch die Codierung der "Richtung" zu erklären. Diese setzt sich aus dx und dy zusammen. Einer der beiden Werte ist null, der andere kann den Wert –2 oder +2 haben. Damit sind die vier Grundrichtungen darstellbar:



Zum Drehen dieses Richtungsvektors in 90-Grad-Schritten werden folgende Formeln benutzt:

vorwärts: dy = +dy und dx = +dx (In BASIC muß dazu eine Hilfsvariable
nach links: dy = +dx und dx = –dy benutzt werden, im Programm immer
nach rechts: dy = –dx und dx = +dy z.)
rückwärts: dy = –dy und dx = –dx

Entsprechend kann man, ohne die "Richtung" zu ändern, mit den folgenden Kombinationen in alle vier Richtungen testen:

```
vorwärts:    TEST (x+dx,y+dy)
nach links:  TEST (x-dy,y+dx)
nach rechts: TEST (x+dy,y–dx)
rückwärts:   TEST (x-dx,y–dy)
```

Diese Angaben sind ein Spezialfall für eine Formel, mit der man Vektoren drehen kann. Normalerweise würde man einen Vektor der Länge 1 wie folgt

mit Sinus und Cosinus in eine bestimmte Richtung w drehen, wobei die Null-Richtung nach oben zeigen soll (wie in LOGO):

dx=l*sin(w)
dy=l*cos(w)

Um den Vektor um den Winkel dw zu drehen, würde man so vorgehen:

dx=l*sin(w+dw)
dy=l*cos(w+dw)

Speziell bei Anwendungen, bei denen der Vektor aber immer um einen konstanten Betrag gedreht wird (beim Füll-Algorithmus immer um 90 Grad oder beim Zeichnen von Kreisen zum Beispiel!), ist es sinnvoll, auf folgende trigonometrische Gleichungen zurückzugreifen:

d.x=dx*cos(dw) – dy*sin(dw)
d.y=dy*cos(dw) + dx*sin(dw)

d.x und d.y sind die Komponenten des neuen Vektors, dx und dy die des alten. Für diese Formel braucht man sin(dw) und cos(dw) nur noch einmal zu berechnen, bzw. bei 90 Grad sind sin(90)=1 und cos(90)=0

Circle

Auf dieser Grundlage läßt sich besonders schnell ein Kreis zeichnen. Zusätzlich muß man sich nur fragen, durch wie viele Streckenzüge der Kreis angenähert werden soll. Wenn man den maximalen Fehler mit einer halben Koordinateneinheit ansetzt, ergibt sich folgende Mindest-Eckenzahl:

n = 1+PI*SQR(radius)

Zu dieser Formel kommt man, wenn man die Differenz zwischen dem Radius und der Höhe in dem gleichseitigen Dreieck bestimmt, das durch zwei Schenkel der Länge des Radius' und dem Schenkelwinkel 2*PI/n betrachtet. Es soll hierauf aber nicht näher eingegangen werden.

Das folgende BASIC-Programm zeichnet jeden beliebigen Kreis:

```
100 x=320:y=200:r=150
110 GOSUB 200
120 GOTO 120
190 '
200 n%=1+PI*SQR(r)
```

```
210 ORIGIN x,y
220 dx!=0:dy!=r:MOVE dx!,dy!
230 s!=SIN(2*PI/n%)
240 c!=COS(2*PI/n%)
250 '
260 FOR n%=1 TO n%
270    z!=dx!*c!-dy!*s!
280   dy!=dy!*c!+dx!*s!
290   dx!=z!
300   DRAW dx!,dy!
310 NEXT
320 RETURN
```

DAS SCREEN PACK

Sowohl die Text- als auch die Grafik-VDU bringen die Grafikinformationen über Linien und Zeichen nicht direkt auf den Bildschirm. Sie greifen beide auf eine untergeordnete Abteilung des Betriebssystems zurück: das Screen Pack. Hier sind eine Unzahl an Routinen versammelt, die mit Farben, Scrollen oder der Codierung von Tinten im Bildschirmspeicher zu tun haben. Hierfür eine separate Abteilung bereitzustellen, ist auch mehr als notwendig. Der Aufbau des CPC-Bildschirms ist nämlich äußerst kompliziert.

Lage des Video-RAMs

Der Bildschirmspeicher belegt immer einen Block des zur Verfügung stehenden RAMs. Beim CPC 6128 kann die zusätzliche RAM-Bank jedoch nicht benutzt werden. Dabei kann das Video-RAM in jeden der vier (normalen) RAM-Blocks gelegt werden. Es ergeben sich nur Einschränkungen durch das Betriebssystem.

Im untersten Block von &0000 bis &3FFF liegt der LOW KERNEL JUMP-BLOCK, und im dritten Speicherviertel von &8000 bis &BFFF liegen die restlichen Jumpblocks. Im allgemeinen ist es nicht sinnvoll, den Bild-Wiederholspeicher hier hineinzulegen. Das geht nur, wenn man auf die Routinen des Betriebssystems komplett verzichtet.

Die Standardlage für das Video-RAM ist der vierte Block, von &C000 bis &FFFF. Es ist aber auch in BASIC möglich, den zweiten Block zu verwenden. Das folgende Programm zeigt ein Beispiel, in dem mit beiden "Bildschirmen" für eine bewegte Grafik gearbeitet wird:

```
 60 ' Grafik-Demo: Animation mit verdecktem Bildaufbau   vs.
 23.5.86   (c) G.W.
 70 ' ------------------------------------------------
'  -----------   --------
```

```
 71 '
 80 MEMORY &3FFF          ' Screen Base-Speicher des Screen Packs:
 81 DEFINT b,p            '          CPC 664/6128: &B7C6
 90 bs=&B7C6              '<-----          CPC 464: &B1CB
 91  b=&40                ' HiByte des Speicherviertels: &4000 oder &C000
 92 p1=&bcff              ' Port-Adresse CRTC: Register
                          ' anwaehlen
 93 p2=&bdff              ' Port-Adresse CRTC: Register
                          ' beschreiben
100 MODE 1:WINDOW 12,28,6,19
110 n=200:l=100           ' Drehschritte und Seitenlaenge des Quadrates
115 w=2*PI/n              ' Drehwinkel
120 s=SIN(w)              ' Sinus    fuer Vektordrehung
130 c=COS(w)              ' Cosinus  fuer Vektordrehung
140 dx=0:dy=l             ' Startvektor: (0,l)
150 ORIGIN 320,200        ' Origin in die Bildschirmmitte
154 '
155 ' DEMO: Drehendes Quadrat
156 ' ----------------------
160 '
170 dz=dx*c-dy*s              ' Den
180 dy=dy*c+dx*s              ' Vektor
190 dx=dz                     ' drehen
200 b=&100-b:                 ' Screen-Basis &40 -> &C0 -> &40
                              ' wechseln
    POKE bs,b                 ' und das Screen Pack darauf
                              ' einstellen
210 CLS:MOVE dx,dy            ' den nicht dargestellten
                              ' Bildschirm loeschen
220 DRAW -dy,+dx:             ' und das Quadrat in den
    DRAW -dx,-dy:             ' nicht dargestellten
    DRAW +dy,-dx:             ' Bildschirmspeicher
    DRAW +dx,+dy              ' zeichnen
230 OUT p1,12:OUT p2,b\4      ' jetzt den Video-Controller auf
                              ' die neue Lage des
240 GOTO 170                  ' Bildschirmspeichers einstellen
```

Die Schleife ab Zeile 170 ist so aufgebaut, daß zunächst das Screen Pack auf den jeweils anderen RAM-Block eingestellt wird und in diesem das neue, leicht gedrehte Quadrat gezeichnet wird, während die Bildausgabe noch aus dem alten Block erfolgt. Erst wenn die Zeichnung beendet ist, wird auch der Video-Controller (CRTC) mit zwei OUT-Befehlen auf den neuen Block umgestellt.

Dadurch ist nicht sichtbar, wie die Grafik aufgebaut wird. Der Unterschied wird deutlich, wenn man das Programm ohne die Zeilen 200 und 230 laufen läßt. Hier sieht man, wie der Bildschirm gelöscht und das neue Quadrat gezeichnet wird.

Wer will, kann auch noch die folgenden Zeile einfügen:

```
225 MOVE 0,0:FILL 1
225 MOVE 0,0:DRAW -dx,dy
```

RAM-Zeilen

Innerhalb eines Speicherblocks ist das RAM in acht "Zeilen" unterteilt, die jeweils 2 kByte (2048 Bytes) umfassen. Jede dieser RAM-Zeilen entspricht einer bestimmten Rasterzeile aller Buchstaben auf dem Bildschirm.

In der ersten RAM-Zeile von &C000 bis &C7FF ist die Grafikinformation für die oberste Zeile der 8*8-Punkte-Matrizen aller Buchstabenpositionen auf dem Bildschirm enthalten.

Das folgende Programm zeigt das:

```
100 FOR i=1 100:PRINT"kjFDGHNBli yh rtgehp";:NEXT
110 FOR i=&C000 TO &C7FF
120   POKE i,255
130 NEXT
```

Die zweite RAM-Zeile von &C800 bis &CFFF ist für die zweite Zeile aller Buchstabenmatrizen zuständig und so weiter bis zur letzten RAM-Zeile von &F800 bis &FFFF, die die untersten Zeilen speichert.

Jede RAM-Zeile gliedert sich in 25 Teilstücke, die den 25 Buchstabenzeilen auf dem Bildschirm entsprechen. Jedes Teilstück, und damit alle 200 Monitorzeilen, ist 80 Bytes lang. Diese Angaben gelten dabei unabhängig vom eingestellten Bildschirmmodus.

Zunächst fällt auf, daß die RAM-Zeilen nicht vollständig ausgenutzt sind. Jede RAM-Zeile ist &800, also 2kByte (2048 Bytes) lang. Benutzt werden aber nur 25*80 (2000 Bytes). Die überzähligen 48 Bytes jeder RAM-Zeile sind nicht benutzt.

Hardware-Scroll

Es ist aber nur sehr bedingt möglich, in diese 8*48 Bytes eigenen Maschinencode zu legen. Im laufenden Betrieb kann sich die Lage der freien Positionen in den RAM-Zeilen nämlich ändern, und zwar dann, wenn der Bildschirm Hardware-mäßig gescrollt wird.

Das Screen Pack hat zwei Möglichkeiten, den Bildschirminhalt nach oben oder unten zu scrollen. Die langsamere ist, die entsprechenden Speicherbereiche innerhalb des Video-RAMs zu verschieben. Das ist leider eine recht zeitintensive Angelegenheit und wird nur benutzt, wenn Windows, die nicht den ganzen Bildschirm umfassen, gescrollt werden sollen.

Soll der gesamte Bildschirm verschoben werden, so kann dies durch Programmieren des Video-Controllers geschehen. Der CRTC bietet nämlich die Möglichkeit, durch Programmieren zweier seiner Register das "erste Byte des Video-RAMs im Speicher" festzulegen. Das wird auch ausgenutzt, um den Bildschirm in ein anderes Speicherviertel zu legen!

Nun ist zunächst einmal zu klären, was passiert, wenn man den Start des Bildschirms beispielsweise auf das 100. Byte der ersten RAM-Zeile festlegt. Dadurch wird der Start der Bildausgabe aus allen anderen RAM-Zeilen auch auf das 100. Byte festgelegt. Alle RAM-Zeilen können nur in gleicher Weise beeinflußt werden, weil der Video-Controller zwischen den RAM-Zeilen nicht mit seinen Speicher-Adreßleitungen sondern den Character-ROM-Adressen unterscheidet. Das ist im Kapitel über das CRTC-IC näher beschrieben.

Daraus wird klar, daß man die Bildausgabe via Hardware nur um ganze RAM-Zeilen-Pakete, also immer 8 Rasterzeilen, auf einmal scrollen kann.

Was passiert nun aber bei der Bildausgabe aus einer RAM-Zeile, deren Anfang auf das 100. Byte festgelegt ist, wenn das Ende erreicht ist?

Zunächst einmal wird die Grafikinformation aus den letzten 48 Bytes benutzt, die vorher (bei einem Offset von Null) noch unbenutzt waren. Aber das langt noch nicht. Es fehlen noch 52 Bytes bis zum Bildschirmende.

2 kByte Speicher lassen sich mit 11 Adreßbits unterscheiden. Das sind die Adreßleitungen MA0 bis MA9 des CRTC, (3 Wörter) und eine, die niederwertigste, wird vom Gate Array geliefert. Wird &7FF inkrementiert, so ergibt das &800. Es kommt zu einem Übertrag auf das nächste Bit, MA10. MA10 und MA11 sind beim Video-Controller im Schneider CPC aber nicht angeschlossen. Deshalb bleibt der Übertrag unberücksichtigt, MA0 bis MA9 enthalten wieder &000, und die Bildausgabe wird am Anfang der jeweiligen RAM-Zeile fortgesetzt.

"Die RAM-Zeilen bilden Ringspeicher."

Das kann man auch bei dem oben wiedergegebenen BASIC-Programm für die erste RAM-Zeile demonstrieren: Vor Start des Programms muß man den Bildschirm nur ein paarmal scrollen, wodurch der Bildschirmstart innerhalb der RAM-Zeilen verschoben wird.

Die durch die POKEs erzeugte Linie startet dann nicht mehr in der linken oberen Ecke, sondern irgendwo innerhalb des Bildschirms. Erreicht das Programm die rechte untere Ecke, so "verschwinden" die nächsten 48 POKEs im nicht benutzten Bereich der RAM-Zeile, bevor die Linie am Bildschirmbeginn weitergezeichnet wird.

Um den Bildschirmspeicher um eine Zeile nach oben oder unten zu scrollen, muß der Bildschirmstart innerhalb der RAM-Zeilen, der sogenannte "Screen Offset", um 80 Bytes verschoben werden. Den CRTC muß man aber nur mit der Hälfte programmieren, weil die niederwertigste Adreßleitung vom Gate Array verwaltet wird.

Verschiebt man den Start einer RAM-Zeile aber nur um zwei Bytes (weniger geht nicht, weil auch hierfür der CRTC nur mit der Hälfte programmiert werden muß!), so wird der Bildschirmspeicher in der Horizontalen gescrollt. Was auf der einen Seite herausrollt, erscheint auf der anderen Seite wieder, allerdings um eine Zeile versetzt. 40 horizontale Scrolls machen einen senkrechten, normalen Scroll aus: 40*2=80.

Das folgende Programm benutzt diesen Effekt, um eine reizvolle Grafik-Demo zu erzeugen:

```
100 MODE 0                          ' Grafik erstellen:
110 FOR i=0 TO 7
120   INK i,i*3:PAPER i:WINDOW i+1,20-i,i+1,25-i:CLS
130 NEXT
140 ofs=0
150 IF INKEY(0)=0 THEN ofs=ofs+40   ' Cursor hoch
160 IF INKEY(2)=0 THEN ofs=ofs-40   ' Cursor runter
170 IF INKEY(8)=0 THEN ofs=ofs+1    ' Cursor links
180 IF INKEY(1)=0 THEN ofs=ofs-1    ' Cursor rechts
190 ofs=(ofs+&400) MOD &400
210 hi=ofs}256+&C0}4                ' MSB der in CRTC-Registern zu
220 lo=ofs AND &FF                  ' LSB programmierenden ersten
                                    '     Speicheradresse
225 CALL &BD19                      ' MC WAIT FLYBACK
230 OUT &BCFF,12:OUT&BDFF,HI        ' MSB programmieren
240 OUT &BCFF,13:OUT&BDFF,LO        ' LSB programmieren
250 GOTO 150
```

Man kann aber auch die beiden Sperrbits MA10 und MA11 des CRTC, die normalerweise immer mit Null programmiert werden, überwinden und von einem Speicherviertel in ein anderes hineinscrollen. Um das zu zeigen, muß man nur die Zeile 190 gegen die folgende austauschen. Dann kann man durch den ganzen Speicher scrollen. Bevor man aber immer in das nächste Speicherviertel kommt, muß man jede Bank viermal durchrotieren, um eben die beiden Bits zu überwinden. Wer auch noch Zeile 99 einfügt, kann überall klar erkennen, wo etwas passiert:

```
99 INK 14,25:INK 15,26
190 ofs=UNT(ofs) AND &7FFF
```

Es gibt aber auch sinnvolle Anwendungen, wie beispielsweise ein schnelles horizontales Scrolling in einer Textverarbeitung. Hier ist es dann wichtig,

nicht nur die Bildausgabe, also den CRTC, zu informieren, sondern auch das Screen Pack. Das geschieht durch Beschreiben des Screen Offset-Speichers im System-RAM des Screen Packs.

```
 60 '  Horizontale Scroll-Demo    vs. 23.5.86        (c) G.W.
 70 ' ------------------------    -----------        --------
 80 os=&B7C4      ' Screen Offset-Speicher: CPC 664/6128: &B7C4,B7C5
 90 '                                       CPC 464:      &B1C9,B1CA
 91 '
100 DIM t$(25)    ' "*** 25 Zeilen Text eingeben: ***"
110 MODE 2:FOR i=1 TO 25:LINE INPUT t$:t$(i)=t$+SPACE$(255-LEN(t$)):NEXT
120 MODE 2:FOR i=1 TO 25:PRINT LEFT$(t$(i),80);:NEXT
140 es=1:ofs=0:f=0
150 IF es<81 AND INKEY(1)=0 THEN f=+2:GOSUB 190 ' Cursor rechts
170 IF es>1  AND INKEY(8)=0 THEN f=-2:GOSUB 190 ' Cursor links
180 f=0:GOTO 150
185 '
186 ' zuerst horizontal scrollen:
187 '
190 ofs=(ofs+f+&800)mod &800   ' Screen Offset neu berechnen
200 POKE os,ofs AND &FF        ' und Screen Pack damit programmieren
210 POKE os+1,ofs\256
220 hi=ofs\2\256+&C0\4         ' Werte fuer den CRTC
230 lo=ofs\2 AND &FF
240 OUT &BCFF,12:OUT&BDFF,HI   ' CRTC programmieren
250 OUT &BCFF,13:OUT&BDFF,LO
260 '
270 ' dann neu entstandene Spalten fuellen:
280 '
290 es=es+f                                ' erste Spalte merken
300 IF f=+2 THEN WINDOW 79,80,1,25:s=es+78   ' rechts fuellen oder
310 IF f=-2 THEN WINDOW  1, 2,1,25:s=es      ' links fuellen
320 CLS:FOR z=1 TO 25:PRINT MID$(t$(z),s,2);:NEXT
330 RETURN
```

Farbzuordnung via Palette

Die Farbausgabe ist beim Schneider CPC zweigeteilt: Im Bildschirm wird zu jedem Pixel die Tintennummer (INK) gespeichert. Erst in dem Moment, in dem das Pixel auf dem Monitor dargestellt wird, wird die Tinte in eine Farbe umgesetzt. Das geschieht in der ULA, die die Ausgabe der Monitorsignale besorgt.

Dazu besitzt die ULA in ihrem Inneren ein paar Register, die mit OUT-Befehlen programmiert werden können. Diese Register werden Color Look Up Table (CLUT) oder speziell beim Schneider CPC "Palette" genannt.

Die Programmierung der Palettenregister ist im Kapitel über die ULA genau beschrieben.

Das Screen Pack unterstützt dabei eine Zuordnung von zwei verschiedenen Farben zu jeder Tinte. Für alle Inks müssen zwei Farben angegeben werden! Bei der Initialisierung des Rechners wird auf dem FRAME FLYBACK TICKER ein EVENT eingehängt (siehe Programmierung eines Software-Interrupts für die Zeit des Strahlhochlaufs im Monitor), das in regelmäßigen Zeitabständen jeder Tinte die eine bzw. die andere Farbe zuordnet. Dabei werden immer alle Farbzuordnungen in der CLUT der ULA neu programmiert, auch wenn nur ein "Wechsel" zwischen zwei gleichen Farben stattfindet.

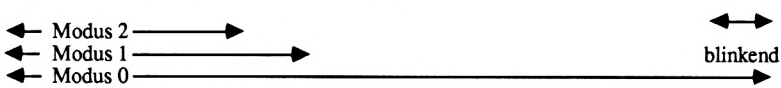
Und das ist der Normalfall: Fast alle Tinten bekommen zwei gleiche Farben zugeordnet, damit sie "nicht" blinken.

In BASIC kann man die Blinkperioden mit SPEED INK periode1,periode2 festlegen. Die Farbzuordnung wird mit dem Befehl INK tinte, farbe1 [,farbe2] vorgenommen.

Dabei muß die ULA mit anderen Farbnummern programmiert werden, als man sie im INK-Statement oder in Maschinensprache bei dem entsprechenden Vektor angeben muß. Das Screen Pack konvertiert die User-Farbnummern erst über eine Tabelle in die "Paletten-Farbnummern". Der Grund für diesen Umstand ist, daß die User-Farbnummern nach ihrer Helligkeit auf einem monochromen Farbmonitor sortiert sind.

Standardzuordnung der Tinten zu Farbnummern:

Tinte:	Border	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Farbe 1:	1	1	24	20	6	26	0	2	8	10	12	14	16	18	22	1	16
Farbe 2:	1	1	24	20	6	26	0	2	8	10	12	14	16	18	22	24	11



Zusammenhang zwischen Farbe, Farbnummer und Paletten-Farbnummer:

Farbe	Paletten-Farbnr.	Farb-nummer	Farb-nummer	Paletten-Farbnr.	Farbe
Schwarz	20	0	26	11	Hellweiß
Blau	4	1	25	3	Pastellgelb
Hellblau	21	2	24	10	Hellgelb
Rot	28	3	23	27	Pastellblaugrün
Magenta	24	4	22	25	Pastellgrün
Hellviolett	29	5	21	26	Limonengrün

Farbe	Paletten-Farbnr.	Farb-nummer	Farb-nummer	Paletten-Farbnr.	Farbe
Hellrot	12	6	20	19	Hellblaugrün
Purpur	5	7	19	2	Seegrün
Hellmagenta	13	8	18	18	Hellgrün
Grün	22	9	17	15	Pastellmagenta
Blaugrün	6	10	16	7	Rosa
Himmelblau	23	11	15	14	Orange
Gelb	30	12	14	31	Pastellblau
Weiß	0	13	13	0	Weiß

Codierung der Tinten

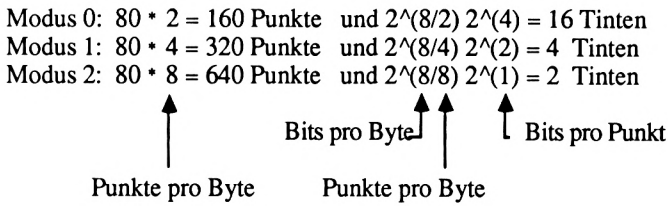
Unabhängig vom eingeschalteten Bildschirmmodus ist ein Byte des Bildschirm-Speichers immer für den gleichen kurzen Abschnitt im Monitorbild zuständig. Je nach Modus wird diese Strecke nur in 2, 4 oder 8 getrennt einfärbbare Pixel unterteilt.

Im Modus 2 wird die "Byte-Strecke" in 8 Pixel unterteilt. Genau ein Bit ist für jedes Pixel zuständig, deshalb kann nur zwischen zwei verschiedenen Tinten unterschieden werden.

Im Modus 1 wird diese Strecke nur in 4 Pixel unterteilt. Deshalb sind jeweils zwei Bits für ein Pixel zuständig, womit sich vier unterschiedliche Tinten unterscheiden lassen.

Im Modus 0 wird das Byte nur noch in zwei Pixel geteilt. Für jedes Pixel können in vier Bits 16 unterschiedliche Tinten codiert werden.

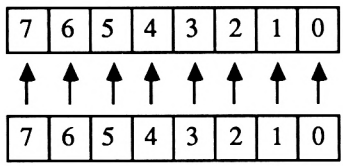
Die folgende Tabelle zeigt noch einmal den Zusammenhang:



Unabhängig vom Modus belegt jeder Buchstabe, der ausgedruckt wird, 8 Pixel in Breite und Höhe. Die Höhe ändert sich nicht, aber die Breite, weil zur Darstellung von 8 verschiedenen Punkten nun ein, zwei oder vier Bytes benötigt werden.

Die Zuordnung der Bits in einem Byte zu den betroffenen Pixeln ist, je nach Modus, äußerst kompliziert. Am einfachsten ist sie noch im Modus 2. Jedes Bit entspricht einem Pixel, das höchstwertige dabei dem Pixel ganz links:

Modus 2:

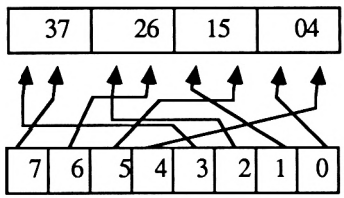


8 Punkte im Bildschirm

Bits 0...7 eines Bytes

Im Modus 1 ist die Sache komplizierter. Wie auch im Modus 0 sind hier nicht zwei nebeneinanderliegende Bits für ein Byte zuständig. Die Bits sind vielmehr nach ihrer Wertigkeit bei der Bestimmung der Tintennummer gruppiert: Die höherwertigen Bits der vier Pixel liegen im unteren Nibble, die niederwertigen Bits im oberen Halbbyte. Innerhalb eines Nibbles folgen die Bits in der Reihenfolge aufeinander, wie die zugehörigen Pixel im Bildschirm.

Modus 2:

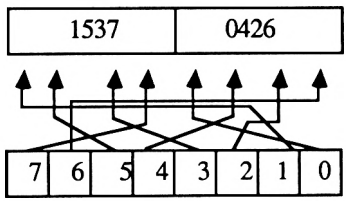


4 Punkte auf dem Bildschirm

Bits 0...7 eines Bytes

Noch komplizierter ist es im Modus 0. Aber die folgende Grafik zeigt, welches Bit bei welchem Pixel mit welcher Wertigkeit in die Bestimmung der Tintennummer eingeht.

Modus 0:



2 Punkte auf dem Bildschirm

Bits 0...7 eines Bytes

Der einzige angenehme Effekt, der sich dadurch ergibt, ist, daß man in jedem Modus ein Byte nur einmal shiften muß, um die Pixel darin um eine Position

zu verschieben. Will man das allerdings für ein horizontales "Soft-Scrolling" benutzen, muß man noch das nicht ganz unbedeutende Problem des Pixel-Übertrags zwischen zwei benachbarten Bytes lösen.

Es ist aber leicht, das n-te Pixel von links in einem Byte ganz nach links zu schieben (n-1mal shiften), um dann, beispielsweise mit dem Vektor &BC2F SCR INK DECODE, daraus die Tintennummer zu berechnen.

Pixel-Masken

Diese komplizierten Zuordnungen sind aber auch für die CPU nicht so leicht zu bewältigen. Deshalb enthält das Screen Pack für jeden Modus eine Tabelle, die das Separieren einzelner Pixel in einem Byte erleichtert.

Diese Tabelle enthält für jedes Pixel, das mit einem Byte darstellbar ist, ein Maskenbyte, mit dem man das einzelne Pixel herausfiltern kann. Da die Anzahl an Pixeln, in die ein Byte unterteilt wird, je nach Modus unterschiedlich ist, ist die Tabelle in jedem Modus auch unterschiedlich lang.

Beim CPC wird die jeweils zutreffende Tabelle mit jedem Moduswechsel ab Adresse &B1CF ins RAM kopiert.

Maskenbytes:

```
Modus 2: Pixel 7 → Byte: &X10000000 = &80 = 128  (links)
         Pixel 6 → Byte: &X01000000 = &40 =  64
         Pixel 5 → Byte: &X00100000 = &20 =  32
         Pixel 4 → Byte: &X00010000 = &10 =  16
         Pixel 3 → Byte: &X00001000 = &08 =   8
         Pixel 2 → Byte: &X00000100 = &04 =   4
         Pixel 1 → Byte: &X00000010 = &02 =   2
         Pixel 0 → Byte: &X00000001 = &01 =   1  (rechts)

Modus 1: Pixel 3 → Byte: &X10001000 = &88 = 136  (links)
         Pixel 2 → Byte: &X01000100 = &44 =  68
         Pixel 1 → Byte: &X00100010 = &22 =  34
         Pixel 0 → Byte: &X00010001 = &11 =  17  (rechts)

Modus 2: Pixel 1 → Byte: &X10101010 = &AA= 170  (links)
         Pixel 0 → Byte: &X01010101 = &55 =  85  (rechts)
```

Farbbytes

Jedesmal wenn man für ein Textfenster oder die Grafikausgabe eine neue Vorder- oder Hintergrund-Tinte festlegt, wird das entsprechende Farbbyte berechnet und in eine Speicherzelle des System-RAMs der Text- bzw. Grafik-VDU kopiert. Dieses Byte ist vollständig mit der gewünschten Tinte "eingefärbt". Würde man dieses Byte mit POKE auf den Bildschirm bringen, so würden alle durch das Byte betroffenen Pixel in dieser Farbe dargestellt.

Codierung der:

Modus 2: Tinte 00 → &X00000000 = &00 = 0
Tinte 01 → &X11111111 = &FF = 255

Mode 1: Tinte 00 → &X00000000 = &00 = 0
Tinte 01 → &X11110000 = &F0 = 240
Tinte 02 → &X00001111 = &0F = 15
Tinte 03 → &X11111111 = &FF = 255

Modus 0: Tinte 00 → &X00000000 = &00 = 0
Tinte 01 → &X11000000 = &C0 = 192
Tinte 02 → &X00001100 = &0C = 12
Tinte 03 → &X11001100 = &CC = 204
Tinte 04 → &X00110000 = &30 = 48
Tinte 05 → &X11110000 = &F0 = 240
Tinte 06 → &X00111100 = &3C = 60
Tinte 07 → &X11111100 = &FC = 252
Tinte 08 → &X00000011 = &03 = 3
Tinte 09 → &X11000011 = &C3 = 195
Tinte 10 → &X00001111 = &0F = 15
Tinte 11 → &X11001111 = &CF = 207
Tinte 12 → &X00110011 = &33 = 51
Tinte 13 → &X11110011 = &F3 = 243
Tinte 14 → &X00111111 = &3F = 63
Tinte 15 → &X11111111 = &FF = 255

Um nun ein Pixel in einer bestimmten Tinte zu setzen, muß man:

- aus Bildschirmbasis und Scroll-Offset und der X- und Y-Koordinate die Adresse des betroffenen Bytes ermitteln;
- mit der X-Koordinate die zutreffende Pixel-Maske bestimmen;
- mit der Maske die nicht betroffenen Bits ausblenden und retten;

- mit der Maske und dem Farbbyte ein Ein-Pixel-Farbbyte erzeugen;
- das Byte mit dem zu setzenden Pixel und das Byte mit den nicht betroffenen Pixeln entsprechend dem Vordergrund-Modus verknüpfen;
- und dieses Byte dann in den Bildschirmspeicher zurücktransportieren.

Nicht gerade wenig, was da für ein einzelnes Pünktchen geleistet werden muß.

Vielfarben-Zeichen

Wenn man im Modus 2 ein Zeichen ausdruckt, so kann man allein mit der Zeichenmatrix alle betroffenen Bits setzen oder löschen, weil sich hier Pixel und Bits 1 zu 1 entsprechen. Macht man die Textausgabe glauben, daß der Modus 2 eingestellt sei, während die Bildausgabe tatsächlich in Modus 1 oder 0 erfolgt, so beeinflussen immer zwei oder vier Bits der Zeichenmatrix ein Pixel auf dem Bildschirm. Man kann dann, einfach indem man entsprechend definierte Sonderzeichen druckt, 4- oder gar 16-farbige Sprites erzeugen. Dabei müssen nur im Farbbyte für PEN immer alle Bits gesetzt sein, damit nicht dadurch wieder Bits aus dem gedruckten Zeichen ausgeblendet werden.

Das folgende Programm definiert ein 4-Farben-Zeichen für Modus 1, das hier 2*2 Buchstabenpositionen groß ist. Um aber 4 Buchstabenpositionen in Modus 1 zu füllen, müssen in Modus 2 acht Zeichen definiert werden. Außerdem muß man mit den Bildschirmgrenzen aufpassen, die je nach Modus verschieden sind.

Das Programm ist so gestaltet, daß man es selbst leicht ändern kann, um für eigene Programme (Spiele, Menüs etc.) Vierfarben-Zeichen zu definieren. Die Auswertung der DATA-Zeilen ist etwas kompliziert geraten, um in den DATA-Zeilen selbst das Zeichen leicht darstellen zu können.

```
 90 ' Vierfarben-Zeichen in Modus 1      vs. 23.5.86     (c) G.W.
 91 '  --------------------------       -----------     --------
 92 '
 93 '   Die einzelnen Zeichen in den Datazeilen bedeuten:
 94 '
100 '    'M' = Ink 3       'X' = Ink 2
110 '    '-' = Ink 1       ' ' = Ink 0
120 '
130 modus=&B7C3   ' Sys-Speicher fuer Modus: CPC 664/6128: &B7C3
140 RESTORE       '                          CPC 464:      &B1C8
150 MODE 1:PEN 3:PAPER 0   ' Modus 1 einstellen; Farbbytes initialisieren
160 POKE modus,2           ' Screen Pack auf Modus 2 einstellen
170 WINDOW 1,80,1,25       ' Fenstergrenzen wieder auf ganzen Bildschirm
180 '
```

```
201 DATA "  MMMMMMMMMMM  "
202 DATA " MMMMMMMMMMMMM "
203 DATA "MMXX          MMM"
204 DATA "MMXXX          MM"
205 DATA "MM XXX----     MM"
206 DATA "MM  XXX----    MM"
207 DATA "MM  -XXX----   MM"
208 DATA "MM  --XXX---   MM"
209 DATA "MM  ---XXX--   MM"
210 DATA "MM  ----XXX-   MM"
211 DATA "MM   ----XXX   MM"
212 DATA "MM    ----XXX MM"
213 DATA "MM          XXXMM"
214 DATA "MMM          XXMM"
215 DATA " MMMMMMMMMMMMM "
216 DATA "  MMMMMMMMMMM  "
220 '
230 DIM s(15,3)              ' DATA-Zeilen auslesen.
240 z$="-XM"
250 FOR i=0 TO 15
260   READ s$
270   FOR j=0 TO 3
280     o$="":u$=""
290     FOR k=1 TO 4
300       b$=BIN$(INSTR(z$,LEFT$(s$,1)),2)
304       o$=o$+RIGHT$(b$,1)
310       u$=u$+LEFT$(b$,1)
320       s$=MID$(s$,2)
330     NEXT
340     s(i,j)=VAL("&X"+o$+u$)
350   NEXT
360 NEXT
370 '
380 z=248                    ' Symbole definieren
390 FOR i=0 TO 8 STEP 8
400   FOR j=0 TO 3
410     PRINT CHR$(25);CHR$(z);
420     FOR k=i TO i+7:PRINT CHR$(s(k,j));:NEXT
430     z=z+1
440   NEXT
450 NEXT                     ' Passende Farben einstellen und Zeichen
                             ' ausgeben
460 '
470 INK 0,0:INK 1,26:INK 2,18:INK 3,15
480 LOCATE 10,10:PRINT CHR$(248);CHR$(249);CHR$(250);CHR$(251)
490 LOCATE 10,11:PRINT CHR$(252);CHR$(253);CHR$(254);CHR$(255)
500 GOTO 500
```

DER KASSETTEN-MANAGER

Die Abteilung, die mit den größten Schwierigkeiten aufwartet, ist wohl der Kassetten-Manager. Der Grund ist, daß man drei Versionen alleine AMSTRAD-intern unterscheiden muß:

- den CASSETTE MANAGER beim CPC 464,
- den CASSETTE MANAGER beim CPC 664 und CPC 6128
- AMSDOS (AMSTRAD Disk Operating System)

Probleme unter BASIC

Dazu kommt noch, daß auch BASIC seine ganz speziellen Probleme mit den Massenspeichern hat.

Sowohl die Disketten- als auch die Kassettenroutinen benötigen für eine Ein- oder Ausgabedatei jeweils einen 2 kByte großen Puffer. Den richtet BASIC immer über HIMEM ein. Weil hier der Speicherplatz aber auch dynamisch an verän- derliche Zeichenmatrizen und Maschinencode-Routinen zugeteilt wird, kann es passieren, daß man plötzlich den Anteil der redefinierbaren Zeichenmatri- zen nicht mehr verändern kann.

Unangenehmer ist jedoch (zumindest beim CPC 464), daß BASIC den Puffer nur bei Bedarf einrichtet und dafür jedesmal eine Garbage Collection im String-Bereich durchführen muß.

Das folgende Programm umgeht das Problem:

```
100 SYMBOL AFTER 256
110 |TAPE:OPENOUT"
120 MEMORY HIMEM-1
130 CLOSEOUT:|DISC
```

Hiernach wird der Puffer nicht mehr freigegeben, und auch die Anzahl der Symbolmatrizen kann wieder frei verändert werden. Wenn beim CPC 464 kein Disketten-Controller angeschlossen ist, dürfen die beiden RSX-Kommandos |TAPE und |DISC natürlich nicht mit eingegeben werden.

Sehr unpraktisch ist, daß man beim CPC 464 das "Breaken" von Kassetten- und Diskettenoperationen nicht unterbinden kann. Das folgende Programm zeigt, wie man beim CPC 664 und 6128 jeden Disketten- oder Kassettenfehler und Breaks des Anwenders abfangen kann. Beim CPC 464 ist das allerdings nicht möglich.

```
5  |TAPE                      ' falls gewuenscht
10 ON BREAK GOSUB 200         ' Break abfangen
15 ON ERROR GOTO 100          ' Fehlerbedingung abfangen
20 SAVE"                ' <---- Bitte mit [ESC] unterbrechen
25 PRINT "zeile 25"
30 GOTO 20
99 '
100 PRINT "Fehler: ";ERR;DERR;" in Zeile ";ERL : RESUME NEXT
199 '
200 PRINT "Break erkannt." : RETURN
```

Probleme in Assembler

Leider hören die Probleme auf der Assemblerebene nicht auf. Die Fehlernummern in DERR sind nämlich nicht nur in BASIC eingeführt worden, sondern auch direkt beim CASSETTE MANAGER des CPC 664/6128. AMSDOS kannte sie sowieso schon.

Dadurch sind jetzt AMSDOS und der CASSETTE MANAGER beim CPC 664/6128 einander noch ähnlicher, als das schon beim CPC 464 der Fall war. Dafür gibt es aber zwischen dem CASSETTE MANAGER des CPC 464 und dem der 6er-Typen Kompatibilitätsprobleme.

Der Fehlerstatus von Unterprogrammen wird beim Schneider CPC gewöhnlich im Carry-Flag zurückgemeldet, wobei ein gesetztes CY-Flag bedeutet, daß die Operation erfolgreich abgeschlossen wurde. So ist es auch bei den Kassetten- und Diskettenroutinen. Eine Ausnahme bildet nur der Katalogvektor.

Tritt aber ein Fehler auf, so fangen die Probleme an: Die Top-Level-Routinen können, bis auf CAS OUT ABANDON und CAS IN ABANDON, einen Fehler zurückmelden. Dabei gibt es zum Teil erhebliche Unterschiede zwischen CAS 464, CAS 664/6128 und AMSDOS.

Die Kassettenoperationen können "gebreakt" werden. Das wird beim CPC 464 laut Firmware-Manual mit CY=0 und Z=1 vermerkt; bei CPC 664 und 6128 mit dem Fehlercode 0 im A-Register. Trotzdem scheinen beide gleich zu arbeiten:

CAS-464, CAS-664/6128: CY=0 und Z=1 und A=0 → Break

Diskettenoperationen können nicht unterbrochen werden.

Alle anderen Fehler werden bei den Kassettenroutinen immer mit CY=0 und Z=0 angemerkt. Der CPC 664/6128 gibt in A zusätzlich aber immer noch einen Fehlercode aus.

AMSDOS kennt noch mehr Fehler, die auch mit CY=0 gemeldet werden. Die Fehlerbedingungen "EOF", "Datei nicht eröffnet" bzw. "Bereits eine Datei eröffnet" werden wie bei den Kassettenroutinen mit Z=0 gemeldet, die anderen aber mit Z=1, was beim Kassettenbetrieb der Break-Meldung entspricht.

Die folgende Tabelle enthält eine Aufstellung der möglichen Fehlercodierungen, wie sie ganz allgemein bei CAS 464, CAS 664/6128 und AMSDOS auftreten können:

Zero-Flag	Fehlernummer im Akku	Bedeutung (A=AMSDOS, 4=CAS 464, 6=CAS664/6128)	
Z=1	0	4 6	Break
Z=0	1 ... 5, 255	4	End of File oder Datei nicht eröffnet oder bereits eine Datei eröffnet
Z=0	14	A 6	Datei nicht eröffnet bzw. bereits eine Datei eröffnet
Z=0	15	A 6	End of File (hard end)
Z=0	26	A	End of File (soft end)
Z=1	16	A	unbekannter Befehl, illegaler Dateiname, bad command
Z=1	17	A	Datei existiert bereits
Z=1	18	A	Datei existiert nicht
Z=1	19	A	Inhaltsverzeichnis ist voll
Z=1	20	A	Diskette ist voll
Z=1	21	A	Diskette wurde bei offener Datei gewechselt
Z=1	22	A	Datei ist schreibgeschützt
Z=1	Bit 6 gesetzt:	A	FDC-Fehler
Z=1	Bit 7 gesetzt:	A	Fehler wurde dem Anwender bereits mitgeteilt

FDC-Fehler: Bit 6 (&40) ist gesetzt. Die folgenden Bits bedeuten:

&01	1	– ID- oder Data-Adreßmarke fehlt	(Diskette ist nicht formatiert)
&02	2	– Diskette ist schreibgeschützt	(Schreibschutzkerbe ist geöffnet)
&04	4	– Sektor nicht auffindbar	(falsches Diskettenformat eingeloggt)
&08	8	– Laufwerk ist nicht bereit	(Diskette nicht/nicht richtig drin)
&10	16	– Pufferüberlauf	(dürfte nie vorkommen)
&20	32	– Kontrollsummen-Fehler	(Diskette möglicherweise beschädigt)

Katalog

Die Fehlerrückmeldung bei CAS CAT weicht stark von diesem allgemeinen Muster ab.

Der Katalog von Kassette muß "gebreakt" werden, weil er sonst für immer andauert. Trotzdem wird das nicht als Fehler mit CY=0 angemerkt. Im Gegensatz dazu muß ein Katalog von Diskette nicht "gebreakt" werden. Dafür merkt

aber jetzt AMSDOS einen Fehler an! Möglicherweise wollte man damit zum Kassettenkatalog kompatibel sein und nahm für dessen RETURN-Bedingung (fälschlicherweise) das Standardverhalten

"Break = Fehler ⇔ CY = 0" an.

Erfolgreiche Kataloge:

CAS CAT	464:	CY=1
CAS CAT	664/6128:	CY=1
DISC CAT	o.k.:	CY=0 und Z=0

Fehler – Input-Stream in use:

CAS CAT	464:	CY=0, Z=0, A=1 (1...5)
CAS CAT	664/6128:	CY=0, Z=0, A=14 = Fehlercode
DISC CAT:		geht: Der Strom wird vorher geschlossen

Fehler – Hardware-Fehler:

CAS CAT: Lesefehler werden im Katalog vermerkt und führen nicht zum Abbruch
DISC CAT: CY=0, Z=1, A=Fehlercode

File Handling

AMSDOS ist so aufgebaut, daß von der Dateiverwaltung her kaum ein Unterschied zum Kassetten-Manager festzustellen ist. Das ist zwar einerseits ein Vorteil, weil so die Kompatibilität erhalten blieb, leider aber ein Nachteil, denn so ist von BASIC aus kein direkter Zugriff auf einzelne Sektoren möglich, und auch relative Dateien mit direktem Schreib/Lese-Zugriff auf einzelne Datensätze wurden nicht implementiert.

Zur gleichen Zeit kann nur eine Datei für Eingabe (Lesen) und eine für Ausgabe (Schreiben) geöffnet werden. Für jede Datei wird ein Puffer von 2 kByte im RAM benötigt.

Das "Handling" ist dabei weitgehend unabhängig davon, ob man eine Ein- oder Ausgabedatei vor sich hat, auf Kassette oder Diskette arbeitet oder in BASIC oder Assembler programmiert:

- Zunächst muß die Datei eröffnet werden, wofür man den Puffer einrichten und den Dateinamen übergeben muß.

- Dann kann in der Datei gearbeitet werden: Entweder liest (oder schreibt) man zeichenweise (das ist in BASIC bei OPENIN, OPENOUT und SAVE "basicprog",a der Fall) oder die ganze Datei auf einmal. Das geht natürlich schneller. Mischen der beiden Möglichkeiten ist aber nicht erlaubt.

- Zum Schluß muß die Datei wieder geschlossen werden.

Bei der Bearbeitung von Diskettendateien von Maschinencode aus muß man zusätzlich noch selbst darauf achten, daß die Diskette eingeloggt ist.

AMSDOS legt in seinem System-RAM nämlich einen "Drive Parameter Block" an, in dem die wichtigsten Formatierungsparameter gespeichert sind. Bezieht sich diese Tabelle beispielsweise noch auf eine Diskette im Datenformat, wenn eine CP/M-Diskette eingelegt wird, so kann AMSDOS diese Diskette nicht beschreiben oder lesen!

Am einfachsten geschieht dies durch Auf- ruf des RSX- Kommandos |A oder |B.

Nicht nur beim CPC 464, auch bei den 6er-Typen sind die Kassettenroutinen vollständig im ROM des Betriebssystems enthalten. Ist auch AMSDOS present, so muß man AMSDOS nur mit |TAPE, |TAPE.IN oder |TAPE.OUT ausblenden.

Kassetten-Meldungen

Die Kassettenroutinen geben bei Bedarf Dialogtexte aus, so zum Beispiel, wenn eine Datei eröffnet wird, um den Kassettenrecorder zu starten. Diese Meldungen können aber auch unterdrückt werden. Dazu muß man den Vektor CAS NOISY aufrufen und im A-Register ein entsprechendes Flag übergeben. BASIC schaltet normalerweise diese Systemmeldungen ein. Wenn man aber einen Dateinamen übergibt, der mit einem Ausrufungszeichen (!) anfängt, so werden die Meldungen von BASIC ausgeschaltet.

Nicht von dieser Regelung betroffen sind die Fehlermeldungen:

Read error a – Lesefehler (unmöglich lange Periode gelesen)
Read error b – Checksummen-Fehler
Read error c – Verify-Fehler
Read error d – Block länger als 2 kBytes

Write error a – geforderte Periodenlänge ist zusammen mit der angegebenen Präkompensation zu kurz.

und die Aufforderung, einen Ladeversuch zu wiederholen:

Rewind tape

Es ist möglich, gleichzeitig eine Ein- und eine Ausgabedatei eröffnet zu haben. Da dann mit zwei verschiedenen Kassetten gearbeitet werden muß, die je nachdem, ob gerade wieder ein Block geladen oder gespeichert werden soll, vom Anwender ausgetauscht werden müssen, sollte man hier immer mit eingeschalteten Meldungen arbeiten.

Header und Data Record

Jede Datei, die vom CASSETTE MANAGER auf Kassette geschrieben wird, wird von ihm in jeweils höchstens 2 kByte lange Blöcke unterteilt.

Jeder Block setzt sich dabei aus zwei "Records" zusammen: einem "Header Record" und einem "Data Record".

Der Header Record enthält 64 Bytes Informationen über den folgenden Data Record, der 1 bis 2048 Bytes lang sein kann und die Daten dieses Blocks enthält.

Aufbau des Header Records:

Byte 00 bis 15:	File-Name
Byte 16:	Blocknummer (von 1 bis ...)
Byte 17:	Flag für den letzten Block einer Datei (<>0 → letzter Block)
Byte 18:	Dateityp (siehe unten)
Byte 19 und 20:	Länge des Data Records in Bytes
Byte 21 und 22:	Quelladresse des Data Records
Byte 23:	Flag für den ersten Block einer Datei (<>0 → erster Block)
Byte 24 und 25:	Gesamtlänge der Datei
Byte 26 und 27:	Startadresse (für Maschinencode-Routinen)
Byte 28 bis 63:	unbenutzt

Dateityp:

Bit 0:	Flag für geschützte Dateien (<>0 → geschützt)
Bit 1 bis 3:	Dateityp: 0 – BASIC-Programm in der internen Darstellung 1 – Speicherabzug (Maschinencode o.ä.)

2 – Bildschirmabzug
3 – ASCII-Datei
4 bis 7: nicht definiert
Bit 4 bis 7: Version: normalerweise immer 0, nur bei ASCII-Dateien 1

Wird eine Datei für Ausgabe eröffnet, so gibt der Vektor CAS OUT OPEN die Adresse des Header-Puffers zurück. Man kann dann den File-Typ, Gesamtlänge und Startadresse beschreiben. Der Typ wird aber schon vorab auf ASCII eingestellt. Soll also eine ASCII-Datei eröffnet werden, muß man in der Regel nichts mehr in den Header eintragen (die Länge ist wahrscheinlich noch unbekannt, und eine Startadresse gibt es auch nicht).

In die unbenutzten Bytes von 28 bis 63 kann das Programm etwas Beliebiges eintragen. Dieser Bereich wird nur jedesmal, wenn eine neue Output-Datei eröffnet wird, standardmäßig mit Nullen beschrieben.

Wird eine Eingabedatei eröffnet, so wird normalerweise so lange von der Kassette gelesen, bis der erste Block der Datei mit dem angegebenen Namen gefunden wird. Dann steht der Header Record des ersten Blocks dem Hauptprogramm sofort zur Verfügung. Gibt man jedoch einen Namen an, der entweder null Zeichen lang ist oder mit einem Null-Byte anfängt, so eröffnet der CASSETTE MANAGER die erste Datei, die er finden kann.

Low Level Cassette Driving

Bei diesem normalen File-Handling handelt es sich um die oberste Ebene des CASSETTE MANAGERs. Man kann aber auch eine Ebene tiefer zugreifen.

Die Vektoren CAS WRITE, CAS READ und CAS CHECK können benutzt werden, um Dateien in einem eigenen Format auf der Kassette abzuspeichern. Dabei steht neben den normalen Funktionen, Speichern und Laden, noch eine dritte bereit, mit der man Daten auf der Kassette mit dem Inhalt des Speichers im Computer vergleichen kann. Damit kann man gerade abgespeicherte Dateien auf Fehler überprüfen.

Diese Vektoren speichern immer einen Record. Die High Level-Routinen rufen jeweils zum Lesen (Speichern) des Header und des Data Records zweimal die entsprechende Low Level-Routine auf.

Die Länge eines Records ist dabei nicht festgelegt, sondern kann jeden beliebigen Betrag von einem Byte bis zu 65536 annehmen.

Synchronisations-Zeichen

Jeder Record erhält beim Speichern auf Kassette eine Kennung, ein Synchronisationszeichen. Soll ein Record wieder gelesen werden, so muß man dieses Zeichen angeben. Nur Records mit diesem Zeichen werden erkannt und geladen.

Die High Level-Routinen benutzen das, um den Header Record vom Data Record zu unterscheiden:

Synchronzeichen: Header Record: &2C
Data Record: &16

Speicherung von 0 und 1

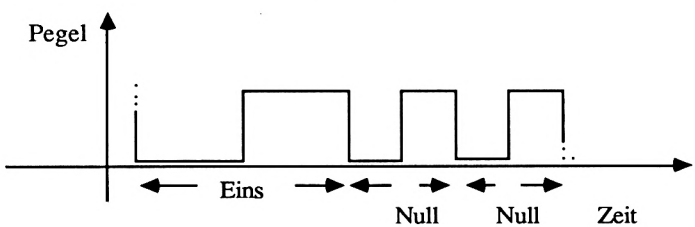
Jeder Record wird auf der Kassette als ein Bitstrom aufgezeichnet. Das heißt, auf der Kassette gibt es keine physikalische Trennung der einzelnen Bytes, Checksummen, Synchronisationszeichen o.ä.

Jedes Bit wird als eine Periode eines Rechtecksignals gespeichert. Ein Rechtecksignal deshalb, weil die Ausgabe nicht analog, sondern digital über ein Bit der PIO erfolgt. Für jedes Bit wird also eine bestimmte Zeit der Datenausgang mit Null und dann mit Eins programmiert.

Dabei ist die Länge des Null- und Eins-Pegels (normalerweise) identisch. Ein Null- und ein Eins-Bit unterscheiden sich in der Länge ihrer Periode: Eins-Bits werden mit der doppelten Periodenlänge abgespeichert.

Das "Timing" wird dabei mit Hilfe des Refresh-Registers des Z80 realisiert. Bei jedem Befehlsholzyklus wird dieses Register erhöht, so daß man hiermit die Anzahl der Befehle zählen kann, die zwischen zwei Flanken des Rechtecksignals abgearbeitet wurden.

Speicherformat von Null- und Eins-Bits:



Daraus ergibt sich auch, daß die Angaben über die Speichergeschwindigkeit, die "Baud-Rate", nur ein Mittelwert sein kann und nur für Dateien zutrifft, die gleich viele Null- und Eins-Bits enthalten.

Man kann also die Speicherung von Daten auf Kassette beschleunigen, indem man Dateien, die besonders viele Eins-Bits enthalten, einfach "invertiert" abspeichert.

Physikalische Probleme

Die analoge Verstärkerelektronik eines Kassettenrecorders neigt dazu, hohe Frequenzen zu unterdrücken. Ein ideales Rechtecksignal hat aber Oberwellen praktisch beliebig hoher Frequenz. Es kommt deshalb zu einer Verrundung des gespeicherten Signals.

Am schlimmsten sind davon die Übergänge von einer kurzen zu einer langen Periode und umgekehrt betroffen, wie es immer bei aufeinanderfolgenden Einsen und Nullen auftritt. Das führt dazu, daß sich der Übergang vom High- zum Low-Pegel so verschiebt, daß der Unterschied zwischen der langen und der kurzen Periode geringer wird. Einsen und Nullen werden schlechter unterscheidbar.

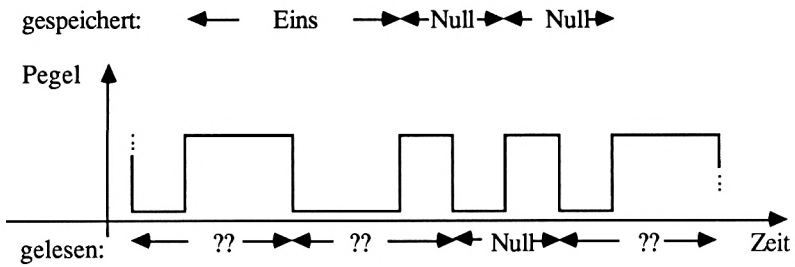
Die Speicher-Software arbeitet deshalb mit einer Vorkompensation, die bei Eins-Null-Übergängen die kurze Periode noch ein wenig kürzer und die lange ein wenig länger macht, so daß sie dann beim Laden möglichst die korrekte Länge haben.

Bei der Programmierung der Speichergeschwindigkeit mit dem Vektor CAS SET SPEED muß man deshalb nicht nur die Länge einer halben Null-Periode, sondern auch die Länge dieser Vorkompensation angeben.

Ein weiteres Phänomen tritt bevorzugt dann auf, wenn man einen externen Kassettenrecorder anschließt: Es kann passieren, daß das Eingangssignal invertiert ist. Das heißt, wo ein Eins-Pegel gespeichert wurde, wird ein Null-Pegel eingelesen, und umgekehrt.

Das muß auf jeden Fall erkannt werden, damit auch immer zwei Periodenhälften, die zum selben Bit gehören, gemessen werden. Würde die Inversion nicht erkannt, so würde immer eine Halbperiode des vorhergehenden Bits mit einer Halbperiode des folgenden zusammen genommen, wodurch die gemessenen Zeiten gemittelt werden. Leider lassen sie sich dann nicht mehr unterscheiden.

Lesefehler bei nicht erkannter Inversion:



Aufbau eines Records

Gegliedert ist ein Record in mehrere Abschnitte: Zunächst einmal kommen &800, also 2048 Eins-Bits, die zur Synchronisation dienen. Normalerweise braucht man dafür aber weit weniger Bits. Beim Schneider CPC bestimmt die Laderoutine aber nach 256 gelesenen Perioden, mit welcher Geschwindigkeit dieser Record gespeichert wurde, und stellt sich automatisch darauf ein.

Danach kommt ein einzelnes Null-Bit, das das Ende des Synchronisationsvorspanns anzeigt. Daran, daß dieses Null-Bit mit einer kurzen Periode auf Null-Pegel oder Eins-Pegel anfing, kann man erkennen, ob das Signal invertiert ist oder nicht.

Das erste Byte danach ist das Synchronisationszeichen, das beim Speichern und Laden immer angegeben werden muß.

Alle Bytes werden mit Bit 7 zuerst abgespeichert. Die Laderoutine überprüft, ob das Sync-Byte übereinstimmt, und setzt nur dann den Ladeversuch fort. Stimmt das Zeichen nicht, so hat die Routine entweder versucht, sich auf eine zufällige Serie aus lauter Eins-Bits zu synchronisieren, oder es handelt sich um einen Record vom falschen Typ (beispielsweise ein Data Record statt eines zu suchenden Header Records).

Erst danach kommen die eigentlichen Daten oder Header-Informationen, wie sie auch immer interpretiert werden. Diese sind, unabhängig von der gespeicherten Anzahl Bytes, immer in 256-Byte-Abschnitte gegliedert. Nach jedem Abschnitt folgen zwei Bytes mit einer Prüfsumme, die nach dem CRC-Verfahren gebildet wird.

Sollen beispielsweise 600 Bytes aus dem RAM gespeichert werden, so ist der Datenteil 3 Abschnitte lang. Der letzte wird mit 3*256–600=168 Null-Bytes aufgefüllt. Nach dem letzten Abschnitt folgen noch 32 Eins-Bits, die vor allem dazu dienen, daß die letzten Datenbytes ohne Verzerrungen abgespeichert werden.

Zur praktischen Anwendung dieser Routinen ist die Kenntnis, wie die einzelnen Bits nun zu Papier (bzw. Magnetband) gebracht werden, natürlich vollkommen uninteressant. Wer aber vielleicht selbst einmal solche Routinen schreiben will als Kopierschutz, oder weil ihm die vorhandenen zu langsam sind, der sieht hieran immerhin, wie es prinzipiell funktionieren muß.

Low Level-Routinen von BASIC aus

Auch unter BASIC kann es sehr interessant sein, auf die Kassetten-Routinen der untersten Ebene zuzugreifen, weil man hier beliebig lange Records abspeichern kann. Damit kann man die Speicher- und Ladezeit verkürzen, ohne mit höheren (und damit unsichereren) Baud-Raten arbeiten zu müssen.

Das folgende Assemblerprogramm bindet einige RSX-Befehle ein, mit denen man beliebige Speicherbereiche speichern, laden und auch überprüfen kann. Außerdem kann man mit |SET.SPEED jede (mögliche) Speichergeschwindigkeit einstellen und die Präkompensation dem eigenen Kassettenrecorder anpassen.

Die Fehlernummern, die die Kassettenroutinen zurückmelden können, werden zwischengespeichert und lassen sich anschließend mit |ERROR,@f% abfragen. Dabei dient der Wert &FF als Kennung für eine ordnungsgemäße Ausführung.

Ein zusätzlicher Fehlercode ist &A0 (=&80 + 32), der AMSDOS-kompatibel gewählt wurde: Code 32 heißt "unbekannter Befehl", was in diesem Zusammenhang auf eine falsche Parameteranzahl hindeuten soll. Bit 7 ist gesetzt, weil der Fehler dem Anwender bereits mitgeteilt wurde, zum Beispiel so:

```
USE: |ERROR,@flag%
```

```
; Low Level-Routinen via RSX     vs.26.5.86          (c) G.Woigk
; --------------------------     ----------          -----------

        ORG  40000

; Deklarationen
```

```
MOTON:   EQU  #BC6E              ; CAS START MOTOR
MOTOFF:  EQU  #BC71              ; CAS STOP MOTOR
LOAD:    EQU  #BCA1              ; CAS READ
SAVE:    EQU  #BC9E              ; CAS WRITE
VERIFY:  EQU  #BCA4              ; CAS CHECK
SPEED:   EQU  #BC68              ; CAS SET SPEED
PCBC:    EQU  #000E              ; LOW KL JP(BC) INSTRUCTION
INTRO:   EQU  #BCD1              ; KL LOG EXT
PRINT:   EQU  #BB5A              ; TXT OUTPUT

; Initialisieren

INIT:    LD   HL,SPACE          ; Zeiger auf 4-Byte-Platz fuer verkettete
Liste
         LD   BC,TABEL          ; Zeiger auf Tabelle externer Kommandos
         JP   INTRO             ; RSX-Paket ins Betriebssystem einbinden

SPACE:   DEFS 4                  ; Platz fuer Hangel-Pointer

TABEL:   DEFW NAMTAB             ; Zeiger --> Namenstabelle
         JP   BSPEED             ; |SET.SPEED
         JP   BLOAD              ; |LOAD
         JP   BSAVE              ; |SAVE
         JP   BVERI              ; |VERIFY
         JP   BERROR             ; |ERROR

NAMTAB:  DEFB "S","E","T",".","S","P","E","E","D"+#80
         DEFB "L","O","A","D"+#80
         DEFB "S","A","V","E"+#80
         DEFB "V","E","R","I","F","Y"+#80
         DEFB "E","R","R","O","R"+#80
         DEFB 0

; Fehlermeldungen:

MSPEED:  DEFM "SET.SPEED, periode, praecomp"
         DEFB 0
MLOAD:   DEFM "LOAD"
         DEFB 0
MSAVE:   DEFM "SAVE"
         DEFB 0
MVERI:   DEFM "VERIFY"
         DEFB 0
MERROR:  DEFM "ERROR, @flag%"
         DEFB 0
MLSV:    DEFM ", sync, adresse, laenge"
         DEFB 0

M1:      DEFB 13,18,10           ; Zeile loeschen und eine Zeile tiefer
         DEFM "  USE: |"         ; Start der Fehlermeldung: "USE:" &
                                 ; RSX-Strich
         DEFB 0

M2:      DEFB 18,13,10,18,0      ; Zeile loeschen, Zeile tiefer, Zeile
                                 ; loeschen
```

```
; Gib Fehlermeldung aus:

ERROR:  EX   DE,HL              ; Zeiger in HL nach DE retten
        LD   HL,M1
        CALL PR$                ; M1 ausdrucken
        EX   DE,HL              ; DE zeigt jetzt auf M2
ER1:    CALL PR$                ; nun Fehlertext ausdrucken
        EX   DE,HL              ; M2 ausdrucken

        LD   A,#80+32           ; vorher Fehler-Flag setzen:
        LD   (FFLAG),A          ; Bit 7=1 --> Fehler bereits mitgeteilt
                                ; Fehler = 32 --> unbekanntes Kommando
                                ;          (hier: falsche Parameterzahl)

PR$:    LD   A,(HL)             ; Zeichen holen
        INC  HL                 ; Zeiger weiterstellen
        AND  A                  ; Zeichen = &00?
        RET  Z                  ; dann Endmarke --> zurueck
        CALL PRINT              ; Zeichen drucken (oder CTRL-Code
                                ; befolgen)
        JR   PR$                ; und naechstes Zeichen

; Fehlermeldung bei LOAD, SAVE UND VERIFY

ERROR2: EX   DE,HL              ; Zeiger nach DE retten
        LD   HL,M1
        CALL PR$                ; M1 ausdrucken
        EX   DE,HL
        CALL PR$                ; Fehlertext ausdrucken
        LD   HL,MLSV            ; Zeiger auf gemeinsamen Text fuer
                                ; Argumente
        JR   ER1                ; so weitermachen, als sei dies der
                                ; Fehlertext
                                ; gewesen
; |SET.SPEED,periode,compens
;
; periode = Laenge eines halben Null-Bits in Mikrosek.:   130...480
; compens = Praekompensation in Mikrosekunden:             0...255

BSPEED: LD   HL,MSPEED          ; Zeiger auf Fehlertext nach HL laden
        CP   2                  ; 2 Argumente?
        JR   NZ,ERROR           ; nein, dann Fehler

        LD   A,(IX+0)           ; A = 1. Parameter = Praekompensation
        LD   L,(IX+2)
        LD   H,(IX+3)           ; HL = 2. Parameter = Laenge 1/2
                                ; Null-Bit
        JP   SPEED              ; CAS SET SPEED

; |SAVE, sync, start, laenge

BSAVE:  LD   HL,MSAVE           ; Zeiger auf Fehlertext
        LD   BC,SAVE            ; Adresse von CAS WRITE
        JR   SLV                ; und weiter
```

```
; |LOAD, sync, start ,laenge

BLOAD:  LD   HL,MLOAD          ; Zeiger auf Fehlertext
        LD   BC,LOAD           ; Adresse von CAS READ
        JR   SLV               ; und weiter

; |VERIFY, sync, start, laenge

BVERI:  LD   HL,MVERI          ; Zeiger auf Fehlertext
        LD   BC,VERIFY         ; Adresse von CAS CHECK
SLV:    CP   3                 ; 3 Argumente?
        JR   NZ,ERROR2         ; nein, dann Fehler

        CALL MOTON             ; CAS START MOTOR
        LD   A,(IX+4)          ; A = 1. Parameter =
                               ; Synchronisationszeichen
        LD   E,(IX+0)
        LD   D,(IX+1)          ; DE = 3. Parameter = Dateilaenge
        LD   L,(IX+2)
        LD   H,(IX+3)          ; HL = 2. Parameter = Startadresse

        CALL PCBC              ; Routine (Adresse in BC) aufrufen

        LD   HL,FFLAG          ; Fehlercode aus A im Fehlerflag
                               ; speichern
        LD   (HL),A            ; Aber Fehler auf &FF setzen fuer 'alles
                               ; o.k.'
        JR   NC,P1             ; falls CY-Flag gesetzt ist
        LD   (HL),#FF

P1:     JP   MOTOFF            ; CAS STOP MOTOR

; |ERROR, @flag%

BERROR: LD   HL,MERROR         ; Zeiger auf Fehlertext
        DEC  A                 ; ein Parameter?
        JP   NZ,ERROR

        LD   L,(IX+0)          ; HL = @flag%
        LD   H,(IX+1)
        LD   A,(FFLAG)         ; Fehler-Flag holen
        LD   (HL),A            ; und in der Integer-Variablen
                               ; abspeichern
        INC  HL
        LD   (HL),0            ; MSB der Variablen nullen
        RET

FFLAG:  DEFB 0                 ; Speicher fuer den Fehlerstatus der
                               ; Routinen

        END
```

AMSDOS

AMSDOS stellt bei den High Level-Routinen fast identische Funktionen bereit. Abgesehen von den Fehlerbedingungen der Vektoren gäbe es keinen weiteren Unterschied, wenn da nicht der Header Record wäre.

Header

Dieser bereitet AMSDOS nämlich ziemliche Schwierigkeiten. Einerseits soll es CP/M-kompatibel sein. Dort kennt man aber keinen Header. Andererseits soll es kompatibel zu den Kassettendateien sein, und die haben einen.

Herausgekommen ist deshalb ein Kompromiß, der leider auch noch fehlerhaft ist.

Zunächst einmal werden alle Nicht-ASCII-Dateien mit einem Header versehen. Dieser ist fast identisch mit dem Kassetten-Header, beansprucht aber einen vollen CP/M-Record mit 128 Bytes. Folgende Bytes sind belegt:

Disketten-Header:

Byte 00: User-Nummer
Byte 01 bis 08: File-Name
Byte 09 bis 11: Extension
Byte 18: Typ-Byte (wie Kas.)
Byte 21 und 22: Originaladresse der Datei, von der sie gespeichert wurde
Byte 24 und 25: Gesamtlänge der Datei
Byte 26 und 27: Einsprungadresse fuer Maschinencode-Programme
Byte 64 und 65: Prüfsumme über Byte 00 bis Byte 66
Byte 67 und 68: (Gesamtlänge der Datei)

Die Prüfsumme ist die Summe aller Bytes von Byte 0 bis Byte 66. Anhand dieser Prüfsumme erkennt AMSDOS beim Laden der Datei mit relativ hoher Wahrscheinlichkeit (aber nicht sicher), daß diese Datei einen Header hat. Der eigentliche Datei-Inhalt beginnt dann erst mit dem zweiten Record.

Demgegenüber haben ASCII-Dateien keinen Header. Das sind unter BASIC alle Dateien, die mit OPENOUT und mit SAVE"basicprog",a gespeichert werden. CP/M-Dateien haben ebenfalls keinen Header Record. Unter CP/M ist das genaue Ende einer COM-Datei unwichtig, weil hier prinzipiell immer ganze Records geladen werden, und die Anzahl der Records wird im Directory-Eintrag vermerkt.

Das exakte Ende einer ASCII-Datei innerhalb des letzten Records erkennt man am Zeichen 26 (&1A), das für diesen Zweck vorgesehen ist.

Normalerweise werden Header-Dateien als Block gespeichert und auch wieder so gelesen; ASCII-Dateien werden meist zeichenweise geschrieben und auch wieder zeichenweise gelesen. Dann gibt es auch keine Probleme.

Der "MERGE-Fehler"

Es ist unter AMSDOS nicht möglich, Headerlose Dateien blockweise zu laden!

Und versucht man, Dateien mit Header zeichenweise zu lesen, so gibt es Schwierigkeiten: Das Zeichen 26 wird, wie das beim zeichenweisen Lesen einer Datei üblich ist, als EOF interpretiert. Dieser Code kann aber auch rein zufällig in der Datei vorkommen, vor allem, wenn sie keinen "normalen" ASCII-Text enthält. Man kann dieses Zeichen aber auch ganz bewußt in eine Datei schreiben, z.B. unter BASIC in eine OPENOUT-Datei: PRINT#9, CHR$(26). Wird dieses Zeichen später eingelesen, so wird es als sogenanntes "weiches" EOF (soft end) interpretiert, auch wenn danach noch 10 kByte Text folgen sollten!

Trotzdem kann man bei einer Datei mit Header erkennen, daß CHR$(26) kein EOF ist, weil im Header ja steht, wie viele Bytes die Datei umfaßt. Bei einer ASCII-Datei dagegen könnte man nur dann ein EOF trotz gelesenem Zeichen 26 sicher ausschließen, wenn das Zeichen in einem anderen als dem letzten Record gefunden wurde.

Soll ein BASIC-Programm zu einem bereits bestehenden in den Speicher gemischt, also mit MERGE angebunden werden, so muß der BASIC-Interpreter das neue Programm zeichenweise einlesen, auch wenn es, wie üblich, blockweise gespeichert wurde. Damit ist der kritische Fall da: Wird jetzt das Zeichen 26 eingelesen, was sehr häufig vorkommen kann (nicht nur bei der Zeilennummer 26), so meldet AMSDOS mitten in einer BASIC-Zeile ein EOF, und der BASIC-Interpreter weiß sich nicht anders zu helfen, als das gesamte Programm zu löschen.

Der einfachste Trick ist, Programme, die später "gemerget" werden sollen, als ASCII-Datei zu speichern:

```
SAVE "progname",a
```

Der von Schneider publizierte Trick besteht in einem Patch des Vektors CAS IN CHAR: Hier wird die Meldung "soft end", also CY=0, Z=0 und A=26 in

"alles o.k.", also CY=1 und A=26, umgewandelt. Damit klappt dann das Mergen von normalen BASIC-Programmen.

Jetzt wird aber beim Mergen von ASCII-Programmdateien das Zeichen 26 nicht mehr als EOF erkannt, und BASIC meldet "Direct command found". Auch INPUT und LINE INPUT aus OPENIN-Dateien schießen jetzt über das Datei-Ende hinaus, weil CHR$(26) nicht mehr als File-Ende erkannt wird. Statt dessen lesen sie, bis mit dem Record-Ende das sogenannte "hard end" erreicht ist.

Trotzdem ist es sicher oft sinnvoll, diesen Vektor zu patchen. Das folgende Assemblerprogramm vermeidet die Schwäche des Schneider-Patches, indem nur bei Header-Dateien der Vektor CAS IN CHAR gepatcht wird. Dateien ohne Header werden nicht gepatcht und liefern deshalb bei CHR$(26) auch weiterhin ein "soft end".

Dieses Beispiel zeigt auch, wie umständlich man die AMSDOS-Vektoren patchen muß, weil diese nicht verschiebbar sind.

Da die beiden RSX-Kommandos |DISC und |DISC.IN die betroffenen Vektoren wieder in den Original-AMSDOS-Zustand versetzen, dürfen sie nicht benutzt werden! Man kann aber selbst zwei RSX-Kommandos mit diesen Namen definieren (die dann Priorität über die "älteren" Definitionen erlangen), die erst die Originalroutinen aufrufen und danach den Patch sofort wieder restaurieren.

```
; Verbesserter CHAIN-MERGE / MERGE-Patch    vs. 25/26.5.86   (c) G.Woigk
; --------------------------------------    --------------   -----------

ORG 40000

INCHAR: EQU  #BC80                ; CAS IN CHAR
OPENIN: EQU  #BC77                ; CAS in OPEN

; Hier  Relocator einbinden

INIT:   LD   HL,OPENIN            ; erstelle eine Kopie (1) des Vektors
        LD   DE,KOPIE1            ; CAS IN OPEN
        LD   BC,3
        LDIR

        LD   HL,INCHAR            ; erstelle eine Kopie (2) des Vektors
        LD   BC,3                 ; CAS IN CHAR
        LDIR

; Hier evtl. noch eigene |DISC- und |DISC.IN-Befehle definieren

INIT1:  LD   HL,OPEN              ; patche den Vektor CAS IN OPEN
```

```
        LD   (OPENIN+1),HL        ; mit Sprung zur eigenen Routine OPEN
        LD   HL,OPENIN
        LD   (HL),195             ; Opcode fuer 'JP'
        RET

KOPIE1: DEFS 3                    ; Platz fuer Original von CAS IN OPEN
KOPIE2: DEFS 3                    ; Platz fuer Original von CAS IN CHAR

; Ersatzroutine fuer CAS IN OPEN

OPEN:   PUSH HL
        LD   HL,(KOPIE1)          ; restauriere Vektor
        LD   (OPENIN),HL          ; CAS IN OPEN
        LD   A,(KOPIE1+2)
        LD   (OPENIN+2),A
        POP  HL

        CALL OPENIN               ; rufe normale Funktion auf

        PUSH HL
        PUSH AF
        CALL INIT1                ; und richte Patch wieder ein

        LD   A,B                  ; logische Dateilaenge = 0?
        OR   C
        CALL NZ,OPEN1             ; Nein -> Datei hat Header, patche CAS
IN CHAR
        CALL Z,OPEN2              ; Ja -> Datei hat keinen Header; Soft
                                  ; EOF darf
        POP  AF                   ; nicht abgefangen werden --> CAS IN
                                  ; CHAR
        POP  HL                   ; restaurieren
        RET

; Patche CAS IN CHAR  (nur HL benutzt wg. INCHR)

OPEN1:  LD   HL,INCHAR            ; Opcode 'JP' in Vektor poken
        LD   (HL),195
        LD   HL,INCHR             ; und Sprungadresse dahinter
        LD   (INCHAR+1),HL
        RET

; Versetze CAS IN CHAR in den Originalzustand

OPEN2:  LD   HL,(KOPIE2)          ; Vektor aus Kopie2
        LD   (INCHAR),HL          ; an seine Position
        LD   A,(KOPIE2+2)         ; im Jumpblock
        LD   (INCHAR+2),A         ; zurueckkopieren
        RET

; Eigene Routine fuer CAS IN CHAR

INCHR:  PUSH HL
        CALL OPEN2                ; CAS IN CHAR restaurieren
        CALL INCHAR               ; CAS IN CHAR aufrufen
```

```
CALL OPEN1              ; CAS IN CHAR wieder patchen
POP  HL

RET  C                  ; CY=1 -> alles o.k.
RET  Z                  ; Fehler mit Z=1 -> kein
                        ; 14/15/26-Fehler
CP   26                 ; Fehler 26 = Soft EOF?
CCF
RET  NC                 ; Fehler < 26: dann zurueck mit CY=0
                        ; und Z=0
CP   27
RET  C                  ; Fehler = 26: dann zurueck mit CY=
                        ; und Z=0
CP   26
RET                     ; Fehler > 26: dann zurueck mit CY=0
                        ; und Z=0
```

Die RSX-Kommandos

Ein Diskettenlaufwerk hat natürlich weit mehr Fähigkeiten als ein Kassettenrekorder. So ist es unter AMSDOS problemlos möglich, eine Ein- und eine Ausgabedatei eröffnet zu halten, und der Zugriff auf einzelne Dateien ist sehr schnell.

Mehr Fähigkeiten bedingen natürlich auch mehr Aufwand, auch für den Anwender. Die zusätzlich benötigten Kommandos sind bei AMSDOS alle mittels RSX-Kommandos eingebunden. Ohne AMSDOS kennt ein nicht weiter ausgebauter CPC 464 zunächst einmal nur einen einzigen Eintrag: den für den BASIC-Interpreter.

Mit AMSDOS kommen noch eine Reihe weiterer dazu. Wie |BASIC handelt es sich bei |CPM um den Aufruf eines Vordergrund-Programms, das die Kontrolle über den Computer übernimmt:

RSX-Aufruf von Vordergrund-Programmen:

|BASIC	Aufruf des BASIC-Interpreters (Kaltstart)
|CPM	Aufruf von CP/M (Kaltstart); es muß eine CP/M-Diskette eingelegt sein

Alle weiteren externen Kommandos stellen Hintergrund-Routinen bereit, die der Auswahl des Ein/Ausgabegerätes und der Datenpflege dienen.

Dokumentierte Befehle von AMSDOS:

|A	Umschalten auf Laufwerk A
|B	Umschalten auf Laufwerk B
|TAPE	Umschalten auf Kassettenbetrieb
|DISC	Umschalten auf Diskettenbetrieb
|TAPE.IN	Umschalten der Eingabe auf Bandbetrieb
|TAPE.OUT	Umschalten der Ausgabe auf Bandbetrieb
|DISC.IN	Umschalten der Eingabe auf Diskettenbetrieb
|DISC.OUT	Umschalten der Ausgabe auf Diskettenbetrieb
|USER,nr	Eingabe einer Benutzernummer (0...15) für den Zugriff auf Disketten. Diese beeinflußt CAT, |DIR, |ERA, |REN und alle Schreib- und Lese-Operationen.
|DIR [,@a$]	Ausgabe eines Inhaltsverzeichnisses. Optional kann mit a$ ein Teil des Disketteninhaltes ausgewählt werden, z.B.: a$="*.BAS":\|DIR,@a$
|DRIVE,@d$	Das in d$ angegebene Laufwerk wird angewählt, z.B.: d$="A":\|DRIVE,@d$
|ERA,@s$	Löschen der in s$ angegebenen Files auf Diskette, z.B.: s$="*.BAK":\|ERA,@s$
|REN,@n$,@a$	Umbenennen eines Files. Der alte Name wird im zweiten Parameter, der neue Name im ersten übergeben, z.B.: a$="Testprog.vs1":n$="Testprog.vs2":|REN,@n$, @a$

Diese Kommandos können auch von einem Maschinencode-Programm aus aufgerufen werden, wofür der Vektor KL FIND COMMAND zum Einsatz kommt. Die Parameterübergabe ist jedoch voll auf "BASIC" zugeschnitten.

Die Übergabe eines Strings ist unter BASIC nur mit der Adresse einer String-Variablen möglich.

Beim CPC 464 muß man dafür die Funktion @ benutzen (@a$), beim CPC 664 und 6128 zwar auch, aber BASIC ist hier etwas intelligenter und übergibt auch bei |ERA,a$ die Adresse von a$.

Das folgende Beispiel zeigt, wie man diese AMSDOS-Kommandos von einem Assembler-Programm aus aufrufen kann:

```
; Aufruf |ERA,@a$ von Mcode aus      vs. 26.5.86    (c) G.Woigk
; ----------------------------      -----------    -----------
;
; Modul-Beschreibung:
;
;     Aufgabe:  Loeschen einer Datei
;
;     Eingabe:  HL zeigt auf den Dateinamen
;               A = Laenge des Namens
;
;     Ausgaben: keine.
;
KLFIND: EQU  #BCD4              ; KL FIND COMMAND
FPCHL:  EQU  #001B              ; LOW KL FAR PCHL
;
ERASE:  LD   (PUFFER+1),HL      ; einen String-Descriptor basteln
        LD   HL,PUFFER          ; zuerst Adresse
        LD   (HL),A             ; und dann Laenge eintragen
        PUSH HL                 ; Adresse des Descriptors auf den Stack
        LD   IX,0               ; und die Adresse des 'letzten
                                ; Arguments'
        ADD  IX,SP              ; nach IX laden (Methode 'BASIC')
        LD   HL,ERA             ; Zeiger auf Name "ERA" einstellen und
                                ; das
        CALL KLFIND             ; Kommando suchen -> HL=Adresse, C=ROM
        LD   A,1                ; A = Anzahl Parameter fuer
                                ; 'Basic-Standard'
        CALL C,FPCHL            ; Routine ab HL in ROM C aufrufen,
                                ; falls
;                               ; Kommando gefunden (CY=1)
        POP  HL                 ; und den Stack wieder korrigieren
        RET                     ; fertig

ERA:    DEFB "E","R","A"+#80    ; RSX-Name
PUFFER: DEFS 3                  ; Platz fuer einen 'String-Descriptor'
```

Diskettenorganisation

Obwohl eine Diskette und eine Kassette sich auf den ersten Blick nicht besonders ähnlich sehen, ist das physikalische Prinzip, mit dem auf den beiden Medien die Daten gespeichert werden, dasselbe: Die einzelnen Bits werden in unterschiedlich lange 0- und 1-Perioden zerlegt, und diese werden magnetisch auf dem Datenträger aufgezeichnet.

Dabei sind die auf der Diskette verwendeten Frequenzen allerdings bedeutend höher, so daß die Daten hier wesentlich schneller zwischen dem Computer und der Diskettenstation übertragen werden.

Außerdem drehen sich Disketten, während der Tonkopf auf derselben Stelle stehenbleibt. Der Tonkopf überstreicht deshalb eine kreisförmige Bahn, einen "Track" auf der magnetisierbaren Scheibe. Mit einem Schrittmotor kann er aber auf verschiedene Kreisbahnen eingestellt werden. Dadurch ist ein äußerst schneller Zugriff auf jeden Punkt der Diskette möglich! Bei den Schneider-Laufwerken werden die Disketten so in 40 Spuren unterteilt. Die Spuren sind dabei von außen her ab 0 durchnumeriert.

Jede Spur muß formatiert werden. Dabei werden Verwaltungsinformationen für das FDC-IC, das die Hauptlast beim Datentransfer trägt, auf der Spur eingezeichnet. Beim Formatieren werden die Spuren in einzelne Kreisausschnitte, die Sektoren, unterteilt. Diese sind unter AMSDOS immer so groß, daß genau 512 Datenbytes in einen Sektor passen. In eine Spur können dabei maximal 9 (mit Tricks auch 10) Sektoren dieser Länge gelegt werden. Jeder Sektor kann dabei, unabhängig von seiner tatsächlichen Lage im Track, eine Nummer von 0 bis 255 erhalten.

Wie die Speicherung einzelner Sektoren genau realisiert ist, ist im Kapitel über den FDC (Floppy Disk Controller) beschrieben. Den Anwender, der nur via Vektoren mit dem AMSDOS kommuniziert, interessiert allenfalls, daß es Sektoren und Tracks gibt.

Formate

Bevor eine leere Diskette benutzt werden kann, muß sie formatiert werden. Damit werden die Sektoren in die Spuren eingezeichnet. Jeder Sektor erhält dabei auch gleich seine Nummer. In eine Spur passen höchstens 9 Sektoren, bei den Nummern hat man aber die freie Auswahl zwischen 256 verschiedenen Werten.

Das hat man sich zunutze gemacht, um drei prinzipiell verschiedene Diskettenformate zu kennzeichnen:

- Beim IBM-Format werden nur 8 Sektoren in jeder Spur ausgenutzt. Diese erhalten Sektornummern von 1 bis 8.

- Disketten im CP/M-Format haben 9 Sektoren pro Spur. Die Sektornummern sind mit einem Offset von &40 markiert, gehen also von &41 bis &49.

- Disketten im Datenformat belegen auch 9 Sektoren pro Spur, haben aber einen Offset von &C0. Die Sektoren sind also von &C1 bis &C9 durchnumeriert.

Die einzelnen Formate unterscheiden sich natürlich nicht nur in den Sektornummern. Das wäre ziemlich sinnlos. Außer der geringeren Anzahl formatierter Sektoren pro Spur beim IBM-Format gibt es noch weitere Unterschiede: Im IBM-Format ist die äußerste Spur als Systemspur reserviert. Im CP/M-Format sind die beiden äußersten Spuren bereits für CP/M vergeben:

Belegung der beiden CP/M-Systemspuren

Spur 0 – Sektor &41:	Boot-Sektor
Spur 0 – Sektor &42:	Konfigurationssektor
Spur 0 – Sektor &43 bis &47:	unbenutzt
Spur 0 – Sektor &48 und 49:	CCP und
Spur 1 – Sektor &41 bis &49:	BIOS

Beim CP/M-Format gibt es noch eine Spielart, das Vendor-Format. Dabei wird die Diskette zwar CP/M-konform formatiert, aber die Systemspuren werden nicht bzw. nicht vollständig beschrieben. Dieses Format dient dazu, CP/M-Programme verkaufen zu können, ohne das Copyright von Digital Research zu verletzen. Der Käufer eines solchen CP/M-Programms muß sich dann einfach seine eigenen CP/M-Systemspuren auf die Diskette kopieren (mit SYSGEN und BOOTGEN).

Damit sind aber auch schon alle Unterschiede zwischen den drei Formaten beschrieben. Das IBM-Format wird wohl am seltensten benutzt (wenn überhaupt), weil sich IBM jetzt auf 3,5-Zoll-Disketten festgelegt hat. Die Möglichkeit, hier Daten und Programme zwischen den Rechnern auszutauschen, ist somit stark eingeschränkt und wiegt den Nachteil, daß man hier viel weniger Daten speichern kann, kaum noch auf.

Datei-Verwaltung

Die Disketten sind so organisiert, daß in der ersten verfügbaren Spur (Track 0 bei Daten, Track 1 bei IBM und Track 2 bei CP/M) das Inhaltsverzeichnis angelegt wird. Es umfaßt 4 Sektoren oder 2 kByte.

Das Inhaltsverzeichnis ist in 64 Einträge, die sogenannten "Extents" zu je 32 Byte unterteilt. Jeder Extent kann maximal eine Datei mit 16 kByte verwalten, bei längeren Dateien werden automatisch mehrere Extents benutzt.

In die Block-Belegungstabelle sind alle Blocks eingetragen, die Daten für diese Datei enthalten. Ein Block umfaßt bei 40 Track/einseitigen Disketten normalerweise immer 1 kByte, so auch bei AMSTRAD. Da die Sektoren aber genau

halb so lang sind, werden immer zwei von ihnen zu einem logischen "Block" zusammengefaßt. Dabei können, bedingt durch die ungerade Anzahl von Sektoren pro Spur, auch zwei Sektoren von zwei aufeinanderfolgenden Tracks zu einem Block zusammengefaßt werden. Die Blocknumerierung startet immer in der Spur mit dem Inhaltsverzeichnis, wobei das Inhaltsverzeichnis selbst die Blöcke 0 und 1 belegt.

Aufbau eines Directory-Eintrags

Bytes	Bedeutung
00	&E5 → nicht belegt (oder gelöscht)
	0...15 → User-Nummer
01 - 08	Name des Files
09 - 11	Extension des Files
09	Bit 7 gesetzt → Datei ist schreibgeschützt
10	Bit 7 gesetzt → Datei wird nicht im Inhaltsverzeichnis angezeigt
12	Nummer des Extents
13 / 14	unbenutzt
15	Länge dieses Extents in Records (128 Bytes)
16 - 31	Block-Belegungstabelle (1 Block = 2 Sektoren = 8 Records)

Die Nummer des Extents wird im Byte 12 festgehalten. Der erste Extent einer Datei hat die Nummer 0. Dateien, die länger als 16 kBytes sind, bekommen einen weiteren Extent zugeordnet, der dann die Nummer 1 erhält usw.

In Byte 15 wird eingetragen, wie viele Records dieser Extent umfaßt. Das stammt noch aus der Zeit der 8-Zoll-Laufwerke. Da hatten alle CP/M-Sektoren eine Länge von 128 Bytes. Als dann aber andere Formate aufkamen, und vor allem, als die Speicherkapazität pro Laufwerk immer weiter stieg, kamen dann nach und nach 256-, 512-, 1024- und 2048-Byte-Sektoren.

CP/M kennt aber nach wie vor nur Sektoren mit 128 Bytes! Diese werden auch als "Record" bezeichnet. Weicht diese logische Sektorlänge von der physikalischen auf der Diskette ab, so ist es Aufgabe des BIOS, die größeren Sektoren in die kleineren Records aufzuteilen.

Bei AMSDOS ergibt sich deshalb folgende Zuordnungen:

1 Block = 2 Sektoren = 8 Records

Die Anzahl der durch einen Extent belegten Records wird in Byte 15 festgehalten. Da ein Extent maximal 16 Blocks verwalten kann, ist der höchste Wert,

den dieses Byte annehmen kann, 16*8 = 128. An diesem Wert erkennt man dann auch, daß "höchstwahrscheinlich" noch ein weiterer Extent folgt, weil die Datei für einen zu lang war. Das muß aber nicht sein, weil die Datei ja auch zufällig genau so lang sein könnte.

Low Level Disk Driving

Nicht nur der CASSETTE MANAGER, auch AMSDOS stellt Vektoren bereit, mit denen man auf einer niedrigeren Ebene Daten von und zur Diskette schikken kann. Dafür wurden von AMSTRAD noch weitere RSX-Kommandos eingerichtet, die aber wegen ihrer Namensgebung und der Art der Parameterübergabe (in Z80-Registern) nicht für einen Einsatz von BASIC aus in Frage kommen.

Die Namen dieser Routinen sind alle ein Byte lang und gehen von CHR 1 bis CHR 9. Da bei allen RSX-Namen das letzte Byte des Namens mit &80 geodert sein muß (gesetztes 7. Bit), ergeben sich die "realen" Namen als CHR &81 bis CHR_&89.

Low Level-Kommandos von AMSDOS:

Name	Funktion	Eingaben
&81	Message ON/OFF	A=0 → ON; A<>0 → OFF
&82	Drive Parameter	HL zeigt auf Tabelle mit div. Zeitkonstanten.
&83	Disk Format Parameter	A=0 → IBM; A=&40 → CP/M; A=&C0 → Daten
&84	Read Sektor	E=Drive, D=Track, C=Sektor, HL=Adr.512Byte-Puffer
&85	Write Sektor	E=Drive, D=Track, C=Sektor, HL=Adr.512Byte-Puffer
&86	Format Track	E=Drive, D=Track, HL=Adr. der Parametertabelle
&87	Seek Track	E=Drive D=Track
&88	Test Drive	A=Drive
&89	Retry Count	A=max. Anzahl für Leseversuche

&81 Message on/off

Mit &81 kann man alle Fehlermeldungen des Disk-Controllers unterdrücken, insbesondere die Frage:

```
Retry, ignore or cancel?
```

Wird in A ein Wert ungleich null übergeben, so fragen die verschiedenen AMSDOS-Routinen nicht mehr, sondern kehren sofort mit einer entsprechenden Fehlermeldung zurück, so als habe der Anwender C für "cancel" eingegeben.

Beim CPC 464 ist beim normalen Diskettenbetrieb von BASIC aus diese Möglichkeit meist nicht einsetzbar, weil dann beim geringsten Fehler das Programm unwiderruflich stehenbleibt. Von dieser Einschränkung sind aber nur die High Level-Routinen betroffen (LOAD, SAVE, OPENIN, OPENOUT etc.). Kataloge und RSX-Kommandos veranlassen den BASIC-Interpreter nicht zum Abbruch!

Man kann also auch beim CPC 464 für die Dauer eines RSX-Kommandos wie |A oder |DIR die Fehlermeldungen abschalten!

Beim CPC 664 und 6128 hat man aber überhaupt keine Probleme. Hier kann man auch die Disk-Breaks abfangen und in der Systemvariablen DERR nachschauen, was vorliegt, um dann selbst entsprechend zu reagieren.

&82 Drive Parameter

Mit &82 kann man alle Wartezeiten, die beim Betrieb mit der Diskettenstation einzuhalten sind, neu festlegen. Normalerweise wird man davon nicht Gebrauch machen, weil die einzelnen Werte eigentlich optimal versorgt sind.

Wer aber Laufwerke eines anderen Anbieters anschließt, kann hiermit vielleicht eine andere Step-Rate einstellen (3 ms statt 12 sind heute eigentlich üblich).

Auch bei Programmen, die oft, aber immer mit kurzen Pausen auf ein Laufwerk zugreifen (Compiler o.ä.), kann man die Nachlaufzeit des Motors erhöhen, damit die Disketten nicht immer kurz vor dem nächsten Zugriff stehen bleiben. Dann muß der Motor nämlich immer neu anlaufen, was zu Verzögerungen führt. Die Standardeinstellung ist:

```
HL →  DEFW 50    ; Wartezeit in 1/50 Sek. bis zum Hochlaufen
                 ; des Motors
      DEFW 250   ; Nachlaufzeit nach einem Zugriff in 1/50
                 ; Sekunden.
      DEFB &AF   ; Wartezeit nach dem Formatieren des
                 ; letzten Sektors
      DEFB &0F   ; Wartezeit beim Spurwechsel in ms.
                 ; zusaetzlich zu:
      DEFB &0C   ; Wartezeit beim Spurwechsel in ms. pro Spur
      DEFB &01   ; Head Unload Time = 32 ms (Wert wie zum
                 ; Prog. des FDC)
      DEFB &03   ; Head Load Time   = 16 ms (Wert wie zum
                 ; Prog. des FDC)
```

&83 Disk Format Parameter

Hat man eine Sektornummer von der Diskette gelesen, so kann man damit gerüstet diesen Vektor aufrufen, um AMSDOS auf das zugehörige Format einzustellen. Dieser Vektor kopiert dann den passenden DISK PARAMETER BLOCK aus dem ROM in den Systemspeicher von AMSDOS.

BASIC nimmt dieses "Login" einer Diskette automatisch vor, bevor es eine Diskettendatei eröffnet. In Maschinencode-Programmen muß man das aber selbst machen!

&84 Read Sektor
&85 Write Sektor

Das sind wohl die interessantesten Vektoren überhaupt. Damit kann man beispielsweise einen Diskettenmonitor oder eine Dateiverwaltung mit wahlfreiem Zugriff auf einzelne Datensätze programmieren. Bei der Sektornummer muß der Format-Offset immer mit angegeben werden! Diese Routinen liefern im CY-Flag ihren Fehlerstatus zurück: CY=1 → o.k., CY=0 → Fehler.

&86 Format Track

Über dieses Kommando kann man eine einzelne Spur komplett formatieren. Vorher sollte man aber mit &83 das gewünschte Format angewählt haben! Also: Akku mit &00, &40 oder &C0 laden und erst &83 aufrufen, weil die Formatierungsroutine folgende Parameter aus dem DISK PARAMETER BLOCK übernimmt:

- Bytes/Sektor
- Sektoren/Track
- Länge der Lücke zwischen Sektor-ID und Daten
- Füllbyte für die leeren Sektoren

Die Parametertabelle, deren Adresse dem Kommando in HL übergeben wird, muß dabei für jeden Sektor 4 Bytes für die Sektor-ID enthalten:

```
HL→ DEFB Spurnummer    (0...39)                  für den
     DEFB Seitennummer (0)                       ersten
     DEFB Sektornummer (&01, &41 oder &C1)       Sektor
     DEFB Sektorgroesse (2)
     ...               ...

     DEFB Spurnummer    (0..39)                  für den
     DEFB Seitennummer (0)                       letzten
     DEFB Sektornummer (&08, &49 oder &C9)       Sektor
     DEFB Sektorgroesse (2)
```

Nach Abschluß des Kommandos ist das CY-Flag gesetzt, wenn alles ordnungsgemäß verlaufen ist.

&87 Seek Track

Mit dem Kommando &87 kann man bereits eine neue Spur anfahren, bevor man darauf Daten liest oder schreibt. Damit kann man bei geschickter Programmierung normalerweise Zeit sparen, weil die Positionierung des Schreib-Lesekopfes vom FDC unabhängig von der CPU vorgenommen wird.

Dieses Kommando ist aber unter diesem Aspekt ungünstig programmiert, weil die CPU hier wartet, bis der FDC fertig ist, um im CY-Flag eine Erfolgs- (CY=1) oder Mißerfolgsmeldung (CY=0) zurückzugeben. Das Kommando versucht es bei einem Mißerfolg zunächst so oft, wie mit &89 angegeben wurde. Erst dann, oder wenn keine Diskette eingelegt ist, wird (evtl., je nach &81) der Anwender gefragt "Retry, ignore, cancel?". Und erst, wenn das alles nichts bringt, kehrt die Routine mit gelöschtem CY-Flag zurück.

&88 Test Drive (fehlerhaft)

Hiermit kann man den Laufwerkstatus (Statusregister 3 des FDC) eines Laufwerks abfragen. Dieses Kommando scheint aber einen dicken Fehler zu enthalten, der die Auswertung der Erfolgsmeldung im CY-Flag und des A-Registers erschwert:

Die meisten FDC-Befehle liefern in ihrer Auswertungsphase als erstes Byte das Statusregister 0. An dessen beiden obersten Bits kann man Erfolg oder Mißerfolg einer Operation erkennen. AMSDOS benutzt zur Auswertung der Resultphase eine Standardroutine, die das CY-Flag nur dann setzt, wenn die beiden obersten Bits des ersten Ergebnis-Bytes null sind. Beim Statusregister 0 bedeutet das: Operation erfolgreich beendet.

Das Kommando &88 testet das Flag und kehrt bei der Meldung "Mißerfolg" (CY=0) mit dem CY-Flag ebenso gesetzt zurück. Ist aber CY=1, so wird das erste Ergebnis-Byte in den Akku geladen und erst dann (mit CY=1) zurückgesprungen.

Nun liefert der FDC-Befehl &04 SENSE DRIVE STATUS als erstes Byte aber nicht das Statusregister 0, sondern bloß einen einzigen Wert: das Statusregister 3.

Die Resultphase-Routine setzt also ihr CY-Flag aufgrund der Bits 6 und 7 des Statusregisters 3. Bit 7 (Fault) ist bei der Beschaltung des Controllers nie gesetzt, Bit 6 zeigt aber WRITE PROTECT an.

Ist die eingelegte Diskette also schreibgeschützt, kehrt &88 immer mit "Mißerfolg" (CY=0) zurück. Ist sie das nicht, meldet das Kommando immer "Erfolg", und nur dann enthält A das Statusregister 3.

Das beste ist, auf diese Rückmeldungen ganz zu verzichten und direkt auf die Bytes der Resultphase zuzugreifen, wie es nach dem noch folgenden letzten Kommando beschrieben ist.

&89 Retry Count

Mit diesem Befehl wird festgelegt, wie viele Versuche AMSDOS bei verschiedenen (vor allem Lese-) Befehlen machen soll, bevor es den Anwender "Retry, ignore or cancel" fragt (bzw. je nach Programmierung mit &81 direkt zurückkehrt).

Der Standardwert ist 10. Älteren Disketten kann man mitunter mit bis zu 256 Leseversuchen "nachhelfen".

Auswertung der Resultphase

Praktisch alle AMSDOS-Routinen, die den FDC programmieren, benutzen zum Auswerten der Resultphase eine Standardroutine, die im AMSDOS-ROM ab Adresse &C91C liegt. Diese Routine liest die bis zu 7 Parameter und legt sie im RAM ab der Adresse &BE4C ab. In Adresse &BE4B wird eingetragen, wie viele Bytes der FDC ausgegeben hat.

Diese "Eigenart" der AMSDOS-Routinen kann man sich zunutze machen und auch von BASIC aus erfragen, wie es "ausgegangen" ist. Die meisten Routinen liefern einen Standard-Parameterblock, der sieben Bytes umfaßt. Davon sind die ersten drei am interessantesten:

&BE4C: Statusregister 0
&BE4D: Statusregister 1
&BE4E: Statusregister 2

Was die einzelnen Bits dieser Register bedeuten, ist im Kapitel über den FDC ausführlich erklärt.

Diskettenmonitor

Das nun folgende BASIC-Programm und die RSX-Erweiterung bilden zusammen einen vollwertigen Diskettenmonitor, mit dem man Sektoren lesen, ändern und wieder auf Diskette schreiben kann.

Das Programm enthält einige Unterprogramme, die sicher auch für andere Anwendungen ganz interessant sind. Es wurde deshalb reichlich mit Kommentaren versehen.

Das Programm ist so gestaltet, daß automatisch das eingelegte Diskettenformat erkannt wird. Man braucht also den Sektornummern-Offset, mit dem der Format-Typ gekennzeichnet wird, nicht mit anzugeben.

Außerdem besteht die Möglichkeit, zu einer Blocknummer (aus dem Inhaltsverzeichnis) sich Spur- und Sektor-Nummern der beiden betroffenen Sektoren ausrechnen zu lassen, wozu die beiden Functions in Zeile 1340/1350 dienen. Man kann sich aber auch mit Hilfe der Cursortasten sektor- oder spurweise durch die Diskette tasten.

Sektor-Inhalte können direkt auf dem Bildschirm geändert werden, wobei man hier volle "Cursor-Freiheit" hat. Die Änderungen werden natürlich zuerst nur im Puffer im RAM des Rechners vorgenommen und müssen zum Schluß mit der Sektor-Schreib-Option auf die Diskette zurückgeschrieben werden. Das ist aber auch gut so, weil man so immer noch die Möglichkeit zu einem Rückzieher hat.

Und vor allem gilt: Wenn nicht ganz gewichtige Gründe dagegen sprechen, immer nur mit einer Kopie arbeiten. Die Gefahr, daß ein Sektor unbrauchbar wird, ist einfach zu groß!

```
; Sektor-Read und -Write      vs. 27.5.86      (c) G.Woigk
; ----------------------      -----------      -----------
;
; Kommandos:       |READ, drive,track,sektor,pufferadresse
;                   |WRITE,drive,track,sektor,pufferadresse
;                  |MESSAGE [,0] --> Meldungen ein,[aus]
;
; Bei der Sektornummer muss der Format-kennzeichnende Offset mit
; angegeben werden
;
        ORG  40000
;
KLFIND: EQU  #BCD4              ; KL FIND COMMAND
INTRO:  EQU  #BCD1              ; KL LOG EXT
;
INIT:   LD   HL,SPACE           ; RSX-Befehle einbinden
        LD   BC,TABEL
        CALL INTRO
;
         LD   HL,RDPUF          ; FAR ADDRESS fuer &84 Read-Sektor bestimmen.
        CALL FIND
         LD   HL,WRPUF          ; FAR ADDRESS fuer &85 Write-Sektor bestimmen.
        CALL FIND
         LD   HL,MESPUF         ; FAR ADDRESS fuer &81 Message ON/OFF bestimmen.
;
```

```
FIND:   PUSH HL             ; KL FIND COMMAND aufrufen, um in Adresse & ROM
        CALL KLFIND         ;   in HL und C zu erhalten
        POP  IX
        RET  NC             ; aber nichts eintragen, falls nicht gefunden
        LD   (IX+1),L
        LD   (IX+2),H       ; FAR ADDRESS basteln
        LD   (IX+3),C
RETURN: RET                 ; fertig
;
RDPUF:  DEFB #84            ; Name: READ TRACK
RDFAR:  DEFW RETURN         ; FAR ADDRESS: Dummy zeigt auf ein Return
        DEFB 0              ; ROM-Status
;
WRPUF:  DEFB #85            ; Name: WRITE TRACK
WRFAR:  DEFW RETURN
        DEFB 0
;
MESPUF: DEFB #81            ; Name: Message ON/OFF
MESFAR: DEFW RETURN
        DEFB 0
;
SPACE:  DEFS 4              ; Platz fuer verkettete Liste der RSX-Kommandos
;
TABEL:  DEFW NAMTAB         ; Zeiger -> Namenstabelle
        JP   READ           ; Vektor -> Routine Read-Sektor
        JP   WRITE          ; Vektor -> Routine Write-Sektor
        JP   MESS           ; Vektor -> Message-Routine
;
NAMTAB: DEFB "R","E","A","D"+#80
        DEFB "W","R","I","T","E"+#80
        DEFB "M","E","S","S","A","G","E"+#80
        DEFB 0
;
; -------------------------
;
MESS:   DEC  A              ; |MESSAGE   --> Meldungen EIN
        CPL                 ; |MESSAGE,0 --> Meldungen AUS
        RST  3*8
        DEFW MESFAR
        RET
;
READ:   LD   HL,RDFAR       ; lade Zeiger auf FAR ADDRESS fuer Read-Sektor
        JR   P1             ; und sonst identisch mit WRITE
;
WRITE:  LD   HL,WRFAR       ; lade Zeiger auf FAR ADDRESS fuer Write-Sector
P1:     LD   (FARPTR),HL    ; Zeiger auf FAR ADDRESS nach RST 3
                            ; eintragen
;
        CP   4              ; pruefe Anzahl der Parameter:
        RET  NZ             ; zurueck, wenn nicht 4
;
        LD   A,255          ; Diskettenmeldungen abschalten
        RST  3*8
        DEFW MESFAR
;
        LD   L,(IX+0)       ; *** Parameter bestimmen: ***
        LD   H,(IX+1)       ; HL = 4. Par. = Adresse des I/O-Puffers
        LD   C,(IX+2)       ;  C = 3. Par. = Sektornummer
        LD   D,(IX+4)       ;  D = 2. Par. = Spurnummer
        LD   E,(IX+6)       ;  E = 1. Par. = Laufwerknummer
```

```
;
        RST  3*8            ; &84 oder &85 aufrufen via FAR CALL
FARPTR: DEFW #0000
;
MESSON: XOR  A              ; Diskettenmeldungen wieder zulassen
        RST  3*8
        DEFW MESFAR
;
        RET                 ; fertig
```

```
1000 DEFINT a-z:GOTO 2550
1001 '
1010 ' +-------------------------------------------------------------------+
1020 ' |  Diskettenmonitor vs. 27.5.86 (c) G.Woigk                         |
1030 ' |  ----------------- ----------- -----------                        |
1040 ' |                                                                   |
1050 ' | **** Achtung: Moeglichst immer mit einer Sicherheitskopie ****    |
1060 ' |              **** der bearbeiteten Diskette arbeiten! ****        |
1070 ' +-------------------------------------------------------------------+
1071 '
1080 ' +------------------------------+------------------------------------+
1090 ' | Auswertung der Resultphase:  |                                    |
1100 ' +------------------------------+                                    |
1110 ' | Ausgabe: f=0 --> alles o.k. sonst f=Fehlercode                    |
1120 ' |          typ, ftrk und spt bestimmt                               |
1130 ' +-------------------------------------------------------------------+
1131 '
1140 f=(PEEK(&BE4C)AND &08)       ' *** Fehlercode wie fuer DERR produzieren: ***
1150 f=f+(PEEK(&BE4D)AND &37)     ' Bit 3 aus Statusreg. 0: Drive not ready
1160 IF f THEN f=f+&40            ' Bits 0,1,2,4,5 aus Statusreg. 1 holen
1170 '                            ' Bit 6: FDC-Fehler falls ein Bit gesetzt
1171 '
1220 ' +-------------------------------------------------------------+----+
1230 ' | Berechnung von Track und Sektornummer der Sektoren eines Blocks: |    |
1240 ' +-----------------------------------------------------------+      |
1250 ' |Eingabe: block = Blocknummer, Resultphase-Bytes des AMSDOS          |
1260 ' |Ausgabe: unterer Sektor: trk0, sek0 = Track und Sektor              |
1270 ' |         oberer  Sektor: trk1, sek1 = Track und Sektor              |
1280 ' |         typ, ftrk und spt                                          |
1290 ' +-------------------------------------------------------------------+
1300 '
1340 DEF FNt(b)=ftrk+ b  \  spt    ' Formel: Track aus 512-Byte-Blocknummer
1350 DEF FNs(b)=1    + b MOD spt   ' Formel: Sektor aus 512-Byte-Blocknummer
1330 '
1180 typ=PEEK(&BE51)AND &C0             ' Sektornummern-Offset (Format-Typ).
1190 spt=9+(typ=&0):sek=MIN(sek,spt)    ' Sektoren pro Track
1191 ftrk = ((typ+&40)AND &FF)\&40      ' Nummer der ersten freien Spur
1360 '
1370 trk0 = FNt(block*2)    : sek0 = FNs(block*2)
1390 trk1 = FNt(block*2+1) : sek1 = FNs(block*2+1)
1400 '
1410 RETURN
1420 '
1430 ' +------------------------+-------------------------------------------+
1440 ' | Login einer Diskette   |                                           |
1450 ' +------------------------+                                           |
```

```
1460 ' | Eingabe: d$ = Laufwerk                                                |
1470 ' | Ausgabe: f=0 -> o.k.   /   f<>0 -> Fehler                             |
1471 ' |          typ                                                          |
1480 ' +-----------------------------------------------------------------------+
1490
1500 PRINT CHR$(21);:LOCATE 1,1    ' Textausgabe abstellen wegen "bad command"
1505 |MESSAGE,0:|DRIVE,@d$         ' Laufwerk anwerfen
1510 PRINT CHR$(6);:|MESSAGE
1515 GOSUB 1140                    ' Resultphase auswerten
1520 IF f=0 THEN RETURN            ' --> o.k.
1530 GOSUB 1630                    ' Fehler behandeln
1540 IF f THEN RETURN              ' --> cancel
1550 GOTO 1500                     ' --> retry
1560 '
1570 ' +---------------------------------+-----------------------------------+
1580 ' | Behandlung von Diskettenfehlern: |                                   |
1590 ' +----------------------------------+                                   |
1600 ' | Ausgabe: f=0 -> retry / f=1 -> cancel                                |
1610 ' +-----------------------------------------------------------------------+
1615 '
1620 GOSUB 1140:IF f=0 THEN RETURN            ' Liegt ueberhaupt ein Fehler vor?
1630 CLS#7:PRINT#7,"Laufwerk ";d$;
1640 IF f AND &35 THEN PRINT#7," Lesefehler,";
1650 IF f AND &02 THEN PRINT#7," schreibgeschuetzt,";
1660 IF f AND &08 THEN PRINT#7," nicht bereit,";
1670 PRINT#7," -- Versuch wiederholen? [J/N]";CHR$(7);
1675 '
1680 i$=UPPER$(INKEY$):IF I$="" THEN 1680
1685 '
1690 IF i$="J" THEN f=0:RETURN
1700 IF i$="N" THEN f=1:RETURN
1710 PRINT#7,CHR$(7);:GOTO 1680
1720 '
1730 ' +------------------------+----------------------------------------------+
1740 ' | Sektor schreiben/lesen |                                              |
1750 ' +------------------------+                                              |
1760 ' | Eingabe: s=1 -> Schreiben /   l=1 -> Lesen                           |
1770 ' | Ausgabe: f=0 -> o.k.      /   f>0 -> Fehler                          |
1780 ' +-----------------------------------------------------------------------+
1790 '
1800 CLS#7:IF l=s THEN RETURN
1810 GOSUB 1500:IF f THEN RETURN                          ' Login
1820 IF l THEN PRINT#7,"Sektor lesen: ";
1830 IF s THEN PRINT#7,"Sektor schreiben: ";
1840 PRINT#7,"Laufwerk ";d$;" / Track";trk;"/ Sektor";sek
1850 PRINT#7,"Optionen: Q-Quit / crsr / T-Track / S-Sektor/ ENTER";
1855 '
1860 i$=UPPER$(INKEY$):IF i$="" THEN 1860 ELSE i=ASC(i$)
1865 '
1880 IF i$="Q"THEN RETURN
1890 IF i=240 AND trk<39 THEN trk=trk+1                   ' Cursor hoch
1900 IF i=241 AND trk>0  THEN trk=trk-1                   ' Cursor runter
1910 IF i=242 THEN sek=sek-1:IF sek=0 THEN trk=trk-1:sek=spt ' Cursor links
1920 IF i=243 THEN sek=sek+1:IF sek>spt THEN trk=trk+1:sek=1 ' Cursor rechts
1930 IF i$="T"THEN INPUT#7," -- Track:",trk
1940 IF i$="S"THEN INPUT#7," -- Sector:",sek
1950 trk=MAX(MIN(trk,39),0)                          ' 0 ... Track ... 39
1960 sek=MAX(MIN(sek,spt),1)                         ' 1 ... Sektor ... spt
1970 IF i<>13 THEN 1810
1980 IF l THEN |READ,ASC(d$)-65,trk,sek+typ,puffer
```

```
1990 IF s THEN |WRITE,ASC(d$)-65,trk,sek+typ,puffer
2000 GOSUB 1140                                        ' Result Phase
2010 IF f THEN GOSUB 1500:IF f=0 THEN 1980             ' Fehler -> Login & Retry
2020 RETURN
2030 '
2040 ' +-----------------------------------------+
2050 ' | Pufferinhalt auf dem Bildschirm ausgeben  |
2060 ' +-----------------------------------------+
2070 '
2080 LOCATE 1,1:LOCATE#1,1,1:LOCATE#2,1,1:ZONE 3
2090 FOR y=basis TO basis+240 STEP 16
2100   PRINT#2,"&";HEX$(y-puffer,3);" >"
2110   FOR x=y TO y+15
2120     PRINT HEX$(PEEK(x),2),
2130     PRINT#1,CHR$(1);CHR$(PEEK(x)AND &7F);
2140   NEXT:PRINT
2150 NEXT:x=0:y=0
2160 RETURN
2170 '
2180 ' +----------------------+-------------------------------------------------+
2190 ' | Pufferinhalt aendern |                                                 |
2200 ' +----------------------+                                                 |
2205 ' | Eingabe: basis = Startadresse des dargestellten Bereichs               |
2210 ' +-----------------------------------------------------------------------+
2220 '
2230 LOCATE x+1,y+1:CALL &BB81                     ' Cursor-Klecks darstellen
2235 '
2240 i$=INKEY$:IF i$="" THEN 2240 ELSE i=ASC(i$)   ' auf Tastendruck warten
2245 '
2250 CALL &BB84                                    ' Cursor-Klecks entfernen
2260 IF i=13 THEN RETURN        ' ENTER -> fertig
2270 IF i=240 THEN y=y-1        ' Cursor hoch
2280 IF i=241 THEN y=y+1        ' Cursor runter
2290 IF i=242 THEN x=x-1        ' Cursor links
2300 IF i=243 THEN x=x+1        ' Cursor rechts
2310 IF x=66 THEN x=50:y=y+1    ' \
2320 IF x<0  THEN x=65:y=y-1    '  \
2330 IF x=47 THEN x=0:y=y+1     '   > Bereichsgrenzen ueberpruefen
2340 IF x=49 THEN x=46          '  /
2350 y=(y+16)MOD 16             ' /
2360 IF i>&EE THEN 2230
2361 '
2365 ' ******* Pufferinhalt aendern *******
2366 '
2370 IF x<48 THEN IF x MOD 3=2 THEN x=x+1:PRINT" ";  ' auf legale Position
2380 IF x=48 OR x=49 THEN 2230                        ' ueberpruefen
2390 IF x>49 THEN 2470
2400 i=VAL("&0"+i$):IF i=0 AND i$<>"0" THEN 2230      ' Cursor steht im Bereich
2410 z=basis+y*16+x\3                                 ' des Hex-Auszugs
2420 IF x MOD 3=0 THEN POKE z,(PEEK(z)AND &F)+i*16
2430 IF x MOD 3=1 THEN POKE z,(PEEK(z)AND &F0)+i
2440 PRINT UPPER$(i$);:LOCATE 51+x\3,y+1
2450 PRINT CHR$(1);CHR$(PEEK(z)AND &7F);
2460 i=243:GOTO 2270
2465 '
2470 z=basis+y*16+x-50:POKE z,i                       ' Cursor steht im Bereich
2480 PRINT CHR$(1);i$;:LOCATE (x-50)*3+1,y+1          ' der ASCII-Darstellung
2490 PRINT HEX$(i,2);:i=243:GOTO 2270
2500 '
2510 ' +----------------------------------+
```

```
2520 ' | *** Start des  Hauptprogramms *** |
2525 ' |     *** Initialisierungen ***     |
2530 ' +-----------------------------------+
2540 '
2550 CLOSEIN:CLOSEOUT
2560 IF HIMEM>30000 THEN MEMORY 29999:LOAD"sector.bin",40000:CALL 40000
2570 puffer=30000:trk=0:d$="A":basis=puffer
2590 PAPER 0:PEN 1:INK 0,0:INK 1,26:MODE 2
2600 PRINT STRING$(240,"#");STRING$(160,"#")
2610 WINDOW 3,78,2,4:CLS:PRINT
2620 PRINT"      KIO - Diskettenmonitor /// vs. 1.0 /// 27.5.86 /// (c) G.Woigk"
2630 WINDOW#7,1,80,7,8
2640 WINDOW#1,58,73,10,25
2650 WINDOW#2,1,7,10,25
2660 WINDOW#0,8,73,10,25
2670 GOSUB 2080:GOSUB 1140              ' Disk-Parameter bestimmen und
2671 '                                  ' Pufferinhalt ausgeben
2672 ' +-------------+
2673 ' | Hauptmenue: |
2674 ' +-------------+
2675 '
2680 CLS#7
2690 PRINT#7,"L = laden / S = speichern / A,B = Laufwerk waehlen";
2700 PRINT#7," / <,> / ?-Block / M = Aendern";
2705 '
2710 i$=UPPER$(INKEY$):IF i$=""THEN 2710 ELSE i=ASC(i$)
2715 '
2720 IF i$="L" THEN s=0:l=1:GOSUB 1800:GOTO 2670
2730 IF i$="S" THEN s=1:l=0:GOSUB 1800:GOTO 2680
2740 IF i$="A"OR i$="B" THEN d$=i$:GOSUB 1500:GOTO 2680
2750 IF i=242 AND basis>puffer THEN basis=puffer:GOTO 2670
2760 IF i=243 AND basis=puffer THEN basis=puffer+256:GOTO 2670
2770 IF i$="M" THEN GOSUB 2220:GOTO 2680
2780 IF i$<>"?"THEN 2680
2781 '
2782 ' *** Laufwerksdaten und Sektoren fuer Block bestimmen ***
2783 '
2790 GOSUB 1500:IF f THEN 2680                      ' Login
2800 CLS#7:PRINT#7,"Laufwerk ";d$;": Format = ";
2810 IF typ=0 THEN PRINT#7,"IBM";
2820 IF typ=&40 THEN PRINT#7,"CP/M";
2830 IF typ=&C0 THEN PRINT#7,"Daten";
2840 INPUT#7," -- Block: ",block
2850 block=MAX(MIN(INT(block),255),0):GOSUB 1310
2860 PRINT#7,"Block"block"= Track"trk0"Sektor"sek0" --  Track"trk1"Sektor"sek1
2870 trk=trk0:sek=sek0:GOTO 2690
```

DER SOUND-MANAGER

Eine weitere eigenständige Abteilung der Firmware ist der Sound-Manager. Die hier zusammengefaßten Routinen ermöglichen es, in besonders komfortabler Weise die Tonausgabe über den PSG (Programmable Sound Generator) zu steuern.

Fähigkeiten

Der Sound-Manager nutzt den Software-Interrupt-Mechanismus des Kernel und hat dafür sogar eine eigene, ansonsten nicht zugängliche "SOUND CHAIN" zugestanden bekommen. 100mal in jeder Sekunde wird diese abgearbeitet und bietet so dem Sound-Manager die Möglichkeit, auch 100mal in jeder Sekunde Frequenz und Amplitude aller drei Kanäle des Geräusch-ICs zu verändern.

Deshalb ist es möglich, weit über die Fähigkeiten des PSG hinaus jeweils 15 eigene Hüllkurven für Frequenz und Amplitude der angeschlagenen Noten festzulegen.

Für jeden Kanal des PSG existiert eine eigene Warteschlange, in der bis zu vier Sound-Befehle zwischengespeichert werden können (eine Queue, die als Ringspeicher in einem Array für Fixed Length Records realisiert ist). Dadurch kann man Wartezeiten, die das "spielende" Programm verursacht, leicht überbrücken.

Um die Tonausgabe vollends von einem Hauptprogramm abzukoppeln, kann man, wieder für jeden Kanal getrennt, einen Software-Interrupt programmieren, der ausgelöst wird, sobald in der Warteschlange des jeweiligen Kanals ein Platz frei wird. Während "im Vordergrund" beispielsweise ein Spiel abläuft und den Spieler in Atem hält, wird "im Hintergrund" per Interrupt das nächste Sound-Statement nachgeschoben, sobald ein Ton abgespielt ist.

Da dabei unter Umständen die Tonerzeugung auf den einzelnen Kanälen mit der Zeit außer Tritt geraten kann, (beispielsweise, weil Achtelnoten auf ganze Hundertstel gerundet werden müssen und nun immer ein klein wenig zu kurz geraten), bietet der Sound-Manager einen ausgefeilten Synchronisationsmechanismus.

Alles in allem ist gerade die Software des Sound-Managers derartig überzeugend gestaltet, daß man es bedauert, daß im Schneider CPC als Hardware "nur" ein AY-3-8912 von General Instruments zum Einsatz kommt.

Das Sound-Statement

Die Programmierung der Tonausgabe unterscheidet sich in Maschinensprache fast überhaupt nicht von BASIC! Die Parameter, die man an einen SOUND-, ENV- oder ENT-Befehl anhängt, werden von BASIC nur ausgewertet und dann mit dem entsprechenden Vektor an den Sound-Manager übergeben. Wer in BASIC Sound-Effekte und Musikstücke programmieren kann, hat es sehr leicht, diese danach in Maschinensprache zu übertragen.

An das SOUND-Statement kann man bis zu sieben Parameter anhängen. In BASIC sind dabei aber nur zwei Pflicht. In Maschinensprache muß man aber immer alle Parameter angeben, und die Reihenfolge ist etwas anders. Die sieben Argumente werden dabei im RAM in einem Parameterblock zusammengestellt, und dem Vektor &BCAA SOUND QUEUE wird im HL-Register ein Zeiger auf diesen Block übergeben.

Was man auch in BASIC immer angeben muß, ist der "Kanalstatus" und die "Periodenlänge" des gewünschten Tones. Beläßt man es bei diesen beiden Parametern, so nimmt BASIC für die Tonlänge als Default 20 Hundertstelsekunden und für die Startamplitude (Lautstärke) den Wert 12 an. Alle restlichen Werte werden auf Null gesetzt.

Die folgende Tabelle zeigt, in welcher Form die einzelnen Werte im Parameterblock aufeinanderfolgen müssen:

```
Parameter der Sound-Anweisung

BASIC: SOUND KS,TP,LEN,SA,AENV,TENV,N

MCode: Vektor &BCAA: HL      ---> Parameterblock

HL ---> DEFB KS      - Kanalstatus für diesen Sound
        DEFB AENV    - gewünschte Amplituden-Hüllkurve
        DEFB TENV    - gewünschte Ton-Hüllkurve
        DEFW TP      - Ton-Periode (entspricht Frequenz)
        DEFB N       - Noise = Rausch-Grundfrequenz
        DEFB SA      - Start-Amplitude (Lautstärke)
        DEFW LEN     - Länge des Tons
```

Der BASIC-Befehl SOUND 7,200 müßte in Assembler durch die folgenden Anweisungen ersetzt werden:

```
        ...
        LD   HL,BLOCK1
        CALL #BCAA
        ...

BLOCK1: DEFB 7          ; Kanalstatus wie angegeben
        DEFB 0          ; Nr. der Default-Amplituden-Hüllkurve (keine Änderung)
        DEFB 0          ; Nr. der Default-Frequenz-Hüllkurve (keine Änderung)
        DEFW 200        ; Ton-Periodenlänge wie angegeben.
        DEFB 0          ; kein Rauschen
        DEFB 12         ; Default-Startamplitude
        DEFW 20         ; Default-Wert für Dauer
```

Kanal-Status

Der Kanalstatus ist dabei Bit-signifikant. Jedes der acht Bits in diesem Byte hat eine eigene Bedeutung:

Bit 0 = 1 → Sound-Befehl gilt für Kanal A
Bit 1 = 1 → Sound-Befehl gilt für Kanal B
Bit 2 = 1 → Sound-Befehl gilt für Kanal C
Bit 3 = 1 → Rendezvous mit Kanal A
Bit 4 = 1 → Rendezvous mit Kanal B
Bit 5 = 1 → Rendezvous mit Kanal C
Bit 6 = 1 → Hold-Status
Bit 7 = 1 → Flush

Mit den ersten drei Bits kann man zunächst einmal bestimmen, für welchen Kanal das Sound-Statement überhaupt gelten soll. Dabei ist es möglich, einen Ton gleichzeitig zu mehreren Kanälen zu schicken, indem einfach mehrere Bits gesetzt werden.

Mit den nächsten drei Bits wird der Rendezvous-Status bestimmt. Hiermit läßt sich die Tonausgabe auf mehreren Kanälen synchronisieren. Soll ein Ton auf dem Kanal X angespielt werden, bei dem ein Rendezvous-Bit für Kanal Y gesetzt ist, so überprüft der Sound-Manager, ob im Kanal Y ein Ton wartet, der seinerseits ein Rendezvous mit X hat; wenn nicht, so muß der Ton im Kanal X erst noch warten, bis sein Partner auch da ist; wenn ja, werden beide jetzt gestartet.

Ein Rendezvous mit sich selbst ist immer erfüllt (wenn man ein Rendezvous mit sich selbst hat, ist man sein eigener Partner). Aber auch Dreierbeziehungen sind möglich: A wartet auf B und C, B wartet auf A und C, und C wartet auf A und B. Erst wenn alle drei Töne mit diesen Bedingungen zur Tonausgabe gelangen, werden sie auch abgespielt.

Übrigens werden bei Sound-Statements, die zu mehreren Kanälen gesandt werden, automatisch die einzelnen Kanäle miteinander synchronisiert:

SOUND &X00000011,... – Ton zu Kanal A und B

ist gleichwertig mit

SOUND &X00001010,... – Ton zu Kanal B und Rendezvous mit A
SOUND &X00010001,... – Ton zu Kanal A und Rendezvous mit B

Diese Synchronisation ist dabei wichtig, weil ja in einem der beiden Kanäle noch Töne in der Warteschlange sein könnten. Würden die Kanäle nicht syn-

chronisiert, dann beginnt der eine Kanal schon, während der andere erst noch auf die Abarbeitung der Töne in seiner Schlange davor warten müßte. Die Handhabung des Sound-Befehls wird durch das Mitdenken des Sound-Managers aber stark vereinfacht.

Ist das Hold-Bit gesetzt, so wird der Ton, sobald er an der Reihe ist, nicht abgespielt. Vielmehr wird dieser Ton so lange festgehalten, bis er mit einem expliziten Kommando freigegeben wird. In BASIC dient dazu der Befehl RELEASE. Dabei muß wieder Bit-signifikant angegeben werden, welche Kanäle "losgelassen" werden sollen.

Speziell für verstopfte Kanäle ist das 7. Bit gedacht. Leichter, als man denkt, hat man nämlich unerfüllbare Rendezvous-Bedingungen programmiert und damit die Tonausgabe blockiert. In diesem Fall hilft ein Ton mit gesetztem Flush-Bit. Ist in einem Sound-Befehl dieses Bit gesetzt, so wartet er nicht, sondern wird sofort abgespielt (außer, wenn gleichzeitig das Hold-Bit gesetzt ist, dann wartet er mit der Tonausgabe, wenn er vorne in der Queue angekommen ist).

Wird dieses Bit als Notbremse gebraucht, ist es am sinnvollsten, den Ton gleichzeitig zu allen drei Kanälen zu schicken. Genau diese Eigenschaft hat auch der CHR$(7)-Piepser. Man kann die Tonausgabe also jederzeit freimachen, indem man dieses Zeichen ausdrucken läßt oder im Direktmodus den Zeileneditor mit einem [CLR] oder [DEL] in einer leeren Zeile zu Warnpiepsern anregt.

Aber auch, wenn Töne direkt auf Ereignisse auf dem Bildschirm reagieren sollen, muß man dieses Bit meist setzen. Hat man beispielsweise den Kanal C dafür gewählt, immer den Sound auszugeben, wenn der Spieler auf den Feuerknopf seines Joysticks drückt, so muß der Ton ausklingen, wenn der Knopf nur einmal gedrückt wird, aber andererseits sofort neu gestartet werden, wenn der Spieler ein zweites Mal drückt:

```
100 ENV 5, 0,15,1, 15,-1,20          ' Volumen-Huellkurve
110 ENT -5, 1,1,1                    ' Frequenz-Huellkurve
120 MODE 1:EVERY 7,3 GOSUB 150       ' ca. 7mal pro Sekunde Feuerknoepfe
                                     ' testen
130 WHILE 1:WEND                     ' und den Rest der Zeit abwarten
140 '
150 IF (JOY(0)AND &30)= 0 THEN RETURN     ' Eine Taste gedrueckt? nein ->
                                          ' Return
160 SOUND &X10000100,50,500,0,5,5,5       ' Tonausgabe
170 BORDER 26                             ' Flash
180 LOCATE 1+INT(40*RND),1+INT(25*RND)
190 PRINT CHR$(238);
200 BORDER 1:RETURN
```

Sollen kompliziertere Ereignisse auf Kommando starten, kann man die Warteschlange eines Kanals mit bis zu vier SOUND-Statements füllen. Wenn im ersten Befehl das Hold-Bit gesetzt ist, wird die Warteschlange nicht abgespielt. Tritt dann das bewußte Ereignis ein, kann man den Kanal abspielen, indem man ihn mit RELEASE freigibt.

Periodenlänge

Mit der Periodenlänge wird die Frequenz des Tons festgelegt. Es handelt sich dabei genau um den Kehrwert der Frequenz, so daß mit kleineren (kürzeren) Perioden größere, höhere Frequenzen erzielt werden.

Diese Periodenlänge ist der Wert, mit dem der PSG programmiert werden muß. Erlaubt sind Werte im Bereich von 0 bis $2^{12}-1=4095$. Wie im Kapitel über den PSG erklärt, wird das Tonsignal durch Teilen des Eingangstaktes von 1 MHz erzeugt. Dieser Takt wird vorab durch 16 und dann durch die angegebene Periodenlänge geteilt.

Das kleinste Raster ist deshalb 16 Mikrosekunden, also 16/1.000.000 Sekunden. Wird der PSG mit der Periodenlänge pl programmiert, so ergibt sich die Frequenz aus der Periodenlänge und umgekehrt wie folgt:

$$\frac{1}{f} = \frac{16 * pl}{1.000.000} \quad \leftrightarrow \quad f = \frac{1.000.000}{16 * pl} \quad pl = \quad \leftrightarrow \quad \frac{1.000.000}{16 * f}$$

Der Kammerton A ist momentan auf 440 Hertz festgelegt. Um ihn wiederzugeben, muß der PSG mit dem folgenden Wert programmiert werden:

$$pl(A) = \frac{1.000.000}{16 * 440} = 142$$

Zwei Töne, die eine Oktave auseinanderliegen, haben ein Frequenzverhältnis von 2:1 zueinander.

In der geläufigen 12-Tonleiter liegen dabei alle Töne um den Faktor $\sqrt[12]{2}$ auseinander. Die Periodenlänge eines Tones, der den Halbtonabstand d zum Kammerton A hat, berechnet sich deshalb wie folgt:

$$pl(A+d) = pl(A) / 2 \uparrow (d/12)$$

Dabei kann d positiv oder negativ sein. Bei d>0 liegt der andere Ton über A und bei d<0 darunter. Wählt man einen Abstand von 12 Halbtönen, so ergibt sich:

pl(A+12) = pl(A) / 2 ↑ (12/12) = pl(A) / 2

So, wie es sein soll, ergibt sich für a der nächsten Oktave (12 Halbtöne Abstand zu A) die halbe Periodenlänge, was der doppelten Frequenz entspricht.

Wer keine Lust hat, die drei- oder vierstelligen Zahlen aus der Tabelle abzutippen, kann die Periodenlängen so jederzeit errechnen! Da aber Exponenten über umfangreiche Polynom-Entwicklungen errechnet werden müssen, wird der CPC dadurch unnötig gebremst, wenn er die Berechnung für jedes SOUND-Statement neu durchführen muß. Am sinnvollsten ist es deshalb, zu Beginn des Programms die benötigten Halbtöne zu berechnen und in einem Integer-Array abzulegen.

Notenlänge

Die Ausgabe der Sound-Befehle wird vom Sound-Manager durch programmierte Software-Interrupts in der SOUND CHAIN gesteuert. Da diese Liste vom Kernel 100mal in jeder Sekunde bearbeitet wird, ist das auch das kleinste Raster, in dem die Tonausgabe beeinflußt werden kann. Die Angabe der Notenlänge wird deshalb immer in Hundertstelsekunden gemacht:

SOUND 2,142,50 → 50/100 = 1/2 Sekunde Kammerton A auf Kanal B

Es ist aber noch eine weitere Variation möglich: Gibt man eine negative Notenlänge –n an, so wird die gewählte (oder Default-) Amplituden-Hüllkurve n-mal abgespielt.

Die Default-Amplituden-Hüllkurve mit der Nummer 0 ist so definiert, daß der Ton 200/100 = 2 Sekunden ohne Änderung gehalten wird. Die folgende Anweisung produziert deshalb einen 6 Sekunden langen Ton:

SOUND 2,142,-3 → 3*2 Sekunden Kammerton A auf Kanal B

Außerdem wird bei einer Notenlänge 0 die Amplituden-Hüllkurve genau einmal abgearbeitet. Dieser Fall entspricht also dem Wert -1.

Startamplitude

Die Lautstärke-Angabe entspricht wieder dem Wert, mit dem der PSG programmiert werden muß. Hierbei sind 16 Werte von 0 bis 15 möglich, wobei

die Amplitude 0 die Tonausgabe abstellt. Der Sound-Manager kann dabei aber mit Werten zwischen 0 und 255 programmiert werden. Er nullt automatisch immer die obersten 4 Bits im übergebenen Byte und zwingt den Wert damit in den gültigen Bereich.

Dabei ist auf die Bezeichnung "Startamplitude" zu achten: Hiermit wird nämlich nur angegeben, mit welcher Lautstärke die Ausgabe des Tons beginnen soll. Die Lautstärke kann danach sofort entsprechend der gewählten (oder Default-) Amplituden-Hüllkurve verändert werden.

Die Default-Hüllkurve 0 ist jedoch nicht änderbar und legt fest, daß der Ton mit einer konstanten Lautstärke ausgegeben werden soll. Damit ist dann die Startamplitude gleich der Dauerlautstärke.

Wird in BASIC keine Amplitude explizit angegeben, so wird als Default der Wert 12 angenommen. Damit wäre dann auch schon die Beschreibung dieses Parameters abgeschlossen, wenn es beim CPC 464 nicht noch eine Besonderheit gäbe:

Benutzt man beim CPC 464 eine Amplituden-Hüllkurve, die die Lautstärke nie ändert (wie das beispielsweise bei der Default-Hüllkurve 0 und bei jeder noch nicht veränderten Hüllkurve der Fall ist), so führt der Sound-Manager hier eine spezielle Interpretation des Lautstärkeparameters durch: Er nimmt automatisch den doppelten Wert an. Der Grund hierfür wird wohl immer ein Geheimnis bleiben, ist so doch nur noch ein groberes Raster mit Werten von 0 bis 7 möglich (die real den Amplituden 0, 2, 4 bis 14 entsprechen).

Die Lautstärke wird erst verdoppelt und dann modulo 16 in den gültigen Bereich gezwungen. Die Default-Lautstärke des BASIC-Interpreters wird deshalb von 12 nach 8 verändert:

$12*2=24 \rightarrow 24 \bmod 16 = 8$

In dieser Beziehung sind CPC 464 und CPC 664/6128 wieder einmal nicht 100 % kompatibel zueinander. Man kann die Probleme aber umgehen, indem man alle Noten nur mit einer definierten Amplituden-Hüllkurve abspielen läßt.

Amplituden-Hüllkurve

Der Sound-Manager bietet die Möglichkeit, neben der unveränderbaren Hüllkurve 0 (konstante Amplitude) 15 eigene Amplituden-Hüllkurven zu definieren. Das ist für jeden Musiker unabdingbar, damit der PSG auch nur annähernd "natürliche" Töne produzieren kann.

Kein Ton, der in der Natur vorkommt, hat bis zum letzten Verklingen eine konstante Lautstärke. Dann würde er nämlich nicht "verklingen", sondern abrupt enden. Vielmehr kann man bei den meisten Geräuschen drei Phasen unterscheiden:

- Attack: Anschwellen (nicht bei "angeschlagenen" Tönen)
- Sustain: Halten
- Delay: Ausklingen

Für jede Hüllkurve können bis zu fünf Abschnitte definiert werden, in denen man jeweils eine bestimmte Anzahl von Schritten, die Schritthöhe (Lautstärkeänderung pro Schritt) und die Schrittlänge (Dauer eines Schrittes) festlegen kann.

Damit kann man man leicht die drei Abschnitte (Attack, Sustain, Delay) nachbilden und, wenn man will, noch ein bißchen mehr. In BASIC benutzt man dazu das ENV-Kommando (ENvelope Volume).

In Maschinensprache geschieht die Programmierung einer Hüllkurve fast genauso: Man muß nur den Vektor &BCBC SOUND AMPL ENVELOPE aufrufen und ihm in A die Nummer der Hüllkurve und in HL den Zeiger auf einen Datenblock übergeben.

Der erste Eintrag im Datenblock ist die Anzahl der angegebenen Hüllkurven-Abschnitte, und danach folgen die Angaben in derselben Reihenfolge wie im ENV-Statement. Dabei müssen nur so viele Abschnitte definiert werden, wie das erste Byte im Parameterblock angibt:

Parameter der Amplituden-Hüllkurven

```
 BASIC: ENV nummer,SZ1,SH1,SL1, ..., SZ5,SH5,SL5
 MCode: Vektor &BCBC: A=Nummer  HL →  Datablock

HL  →  DEFB ANZ     - Anzahl Hüllkurven-Abschnitte
       DEFB SZ1
       DEFB SH1     } Parameter des ersten Abschnittes
       DEFB SL1
        ...          ...
        DEFB SZ5     -Schrittzahl
        DEFB SH5     -Schritthöhe
        DEFB SL5     -Schrittlänge
```

Folgende Werte sind erlaubt:

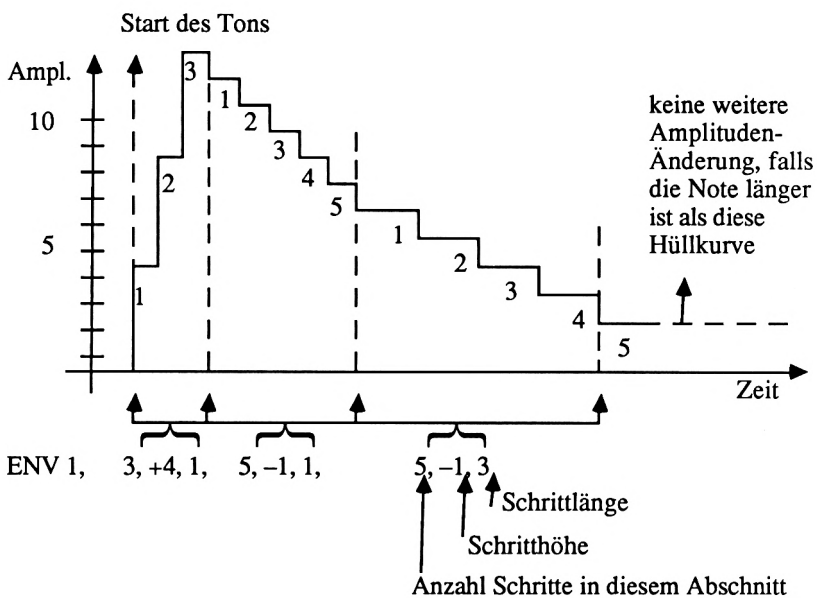
Hüllkurven-Abschnitt (Software-kontrolliert)

- Schrittzahl: 0 ... 127 (0 setzt die Amplitude absolut auf die Schnitthöhe)
- Schritthöhe: –128 ... 127 (entspricht 0 ... 255 in Assembler)
- Schrittlänge: 0 ... 255 (wobei 0 einer Länge von 256 entspricht)

Um eine Hüllkurve zu programmieren, muß man den gewünschten Verlauf der Amplitude in einem Diagramm eintragen und in bis zu 5 verschiedene Abschnitte unterteilen.

Jeden Abschnitt muß man dann in eine geeignete Anzahl Treppenstufen unterteilen, die alle die gleiche Länge und Höhe haben.

Das folgende Diagramm ist ein Beispiel:



Die Diagramme im Handbuch des CPC 464, mit denen die Zerlegung eines Notenabschnittes in diese Treppenstufen für das ENV-Kommandos demonstriert werden, sind dabei nicht ganz korrekt: Hier sehen die einzelnen Trep-

penstufen so aus, als würde jeweils erst gewartet und dann das Volumen verändert. In Wirklichkeit ist aber genau andersrum: Zuerst wird das Volumen verändert und dann gewartet.

Diese Hüllkurve besteht aus drei Abschnitten. Der erste, Attack, ist drei Hundertstelsekunden lang und steigert die Lautstärke um 3*4=12 Werte. Daraus ergibt sich, daß diese Hüllkurve mit einer Startamplitude im Bereich von 0 bis 3 benutzt werden kann! Größere Startamplituden bedeuten nämlich, daß die Lautstärken über den erlaubten Bereich (0...15) hinausgehen und modulo 16 wieder bei 0 beginnen.

Würde die Startamplitude beispielsweise mit 5 festgesetzt, so würde sie wie folgt geändert:

```
-  5+4  =  9  mod 16 =  9  (1. Hundertstelsekunde)
-  9+4  = 13  mod 16 = 13  (2. Hundertstelsekunde)
- 13+4  = 17  mod 16 =  1  (3. Hundertstelsekunde) ← Überschreitung des
-  1-1  =  0  mod 16 =  0  (4. Hundertstelsekunde) ← zulässigen Bereichs
-  0-1  = -1  mod 16 = 15  (5. Hundertstelsekunde)
- 15-1  = 14  mod 16 = 14  (6. Hundertstelsekunde)
```

Einige weitere Besonderheiten müssen geklärt werden. Dauert eine angeschlagene Note länger, als die benutzte Hüllkurve definiert ist, so wird der zuletzt erreichte Wert ausgehalten. Ist die Note kürzer, so wird die Hüllkurve abgebrochen.

Wird in einem Bereich der Hüllkurve eine Schrittzahl mit 0 Schritten festgelegt, so wird die zugehörige Schritthöhe als absoluter Wert betrachtet und die Lautstärke auf diesen Wert gesetzt. Macht man von dieser Möglichkeit im ersten Abschnitt Gebrauch, so wird die Angabe einer Startamplitude im SOUND-Statement praktisch wirkungslos. Weil die Amplitude *vor* der Wartezeit auf den neuen Wert gesetzt wird, kommt die Startamplitude keine einzige Hundertstelsekunde zum Zug. Wird die Lautstärke auf einen absoluten Betrag gesetzt, so wird die angegebene Schrittlänge einmal abgewartet.

Bei den Schrittlängen wird die Angabe 0 als 256/100 Sekunden interpretiert.

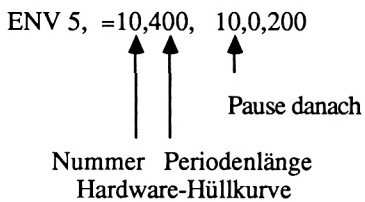
Hardware-Hüllkurven

Außerdem ist es möglich, den Hüllkurven-Generator des PSG selbst zu programmieren. Das kann innerhalb eines ENV-Befehls geschehen. Dazu wird nur das Layout der drei Bytes eines Abschnittes etwas verändert.

Wird in einem Abschnitt eine PSG-Hüllkurve definiert, so werden die Register 11, 12 und 13 des PSG damit programmiert: Hüllkurven-Form und Länge der Veränderungsperiode. Danach wird sofort (ohne auch nur eine Hundertstelsekunde zu warten) der nächste Abschnitt der Hüllkurve bearbeitet.

Dieser sollte deshalb eine Pause sein (Schritthöhe = 0), die so lang definiert werden muß, wie die Hardware-Hüllkurve abgearbeitet werden soll. Folgt kein weiterer Abschnitt, so arbeitet der Sound-Manager automatisch die Default-Amplituden-Hüllkurve 0 ab, in der eine Wartezeit von 2 Sekunden definiert ist.

Die Programmierung der Hardware-Hüllkurven ist zwar in allen drei CPC-Handbüchern erwähnt. In keinem einzigen steht aber, wie man es machen muß. Ein Hardware-Hüllkurven-Abschnitt muß in BASIC mit einem Gleichheitszeichen (=) gekennzeichnet werden:



In Maschinensprache wird ein solcher Abschnitt dadurch gekennzeichnet, daß das 7. Bit der Schrittzahl gesetzt ist. Für Software-Abschnitte ist ja nur eine Schrittzahl von 0 bis 127 erlaubt, also genau die Werte, bei denen das 7. Bit noch nicht gesetzt ist.

Das 7. Bit wird dann ausmaskiert, und Register 13 des PSG (Hüllkurven-Nummer) wird damit beschrieben.

Die folgenden beiden Bytes (normalerweise Schritthöhe und -länge) werden nicht mehr getrennt betrachtet, sondern zusammen als ein Word. Dieser Wert wird dann in die Periodenlängen-Register 11 und 12 des PSG geschrieben (wieder getrennt: MSB und LSB des Wortes in je ein Register).

```
; Amplituden-Huellkurve (PSG-kontrolliert)
; ----------------------------------------

DEFB #80+HKN      ; Huellkurven-Nummer
DEFW LWP          ; Laenge der Wiederholungsperiode
```

Frequenz-Hüllkurven

Neben den 16 Software-Volumen-Hüllkurven gibt es auch noch 16 fast völlig entsprechend aufgebaute Frequenz-Hüllkurven.

Die nicht veränderliche Hüllkurve 0, die von BASIC auch immer als Default angenommen wird, erzeugt einen Ton konstanter Frequenz.

Bei den 15 selbst definierbaren Frequenz-Hüllkurven kann man wieder insgesamt 5 Abschnitte definieren, für die man jeweils Schrittzahl, Schritthöhe und Schrittdauer angeben muß.

Die Programmierung dieser Hüllkurven geschieht dabei in Maschinensprache wieder auf fast identische Weise:

Parameter der Tonperioden-Hüllkurven

```
BASIC: ENT (-)nummer,SZ1,SH1,SL1, ...,  SZ5,SH5,SL5
MCode: Vektor &BCBF:  A=Nummer   HL →   Datablock

HL   →    DEFB ANZ     = Anzahl Hüllkurven-Abschnitte
                          Bit 7 = 1 → wiederholend
          DEFB SZ1
          DEFB SH1     } Parameter des ersten Abschnittes
          DEFB SL1
          ...              ...
          DEFB SZ5       - Schrittzahl
          DEFB SH5       -Schritthöhe (Änderung der Periodenlänge)
          DEFB SL5       - Schrittlänge (Dauer)
```

Für die Parameter der einzelnen Abschnitte gelten wieder ähnliche Grenzen:

- Schrittzahl: 0 ... 239 (0 entspricht 1)
- Schrittweite: –128 ... +127
- Schrittlänge: 0 ... 255 (0 entspricht wieder 256)

Das Ganze funktioniert wie beim ENV-Kommando, nur daß sich die Änderungen jetzt auf die Periodenlänge, also die Frequenz der ausgegebenen Note, beziehen.

Ist bei der Programmierung in Maschinensprache im ersten Byte (Anzahl) das 7. Bit gesetzt, so wird damit eine Hüllkurve definiert, die sich bei Bedarf beliebig oft wiederholen kann. Ist die Frequenz-Hüllkurve fertig abgespielt, bevor

ein Ton komplett abgespielt ist, so wird sie von vorne wiederholt. In BASIC wird das mit einer negativen Hüllkurven-Nummer festgelegt:

```
ENT -5, ....
```

Ist dieses Bit nicht gesetzt, so wird nach Ablauf der Hüllkurve die Frequenz des Tons nicht mehr geändert.

Ungünstig, aber ohne unverhältnismäßig hohen Aufwand nicht zu umgehen, ist dabei, daß die Änderung der Periodenlänge linear ist, während die einzelnen Noten einer Oktave in einem logarithmischen Raster aufeinanderfolgen.

Will man ein Vibrato programmieren, so muß man praktisch für alle 3 bis 4 Halbtöne eine neue Hüllkurve definieren, in denen die Änderungen der Periodenlänge mit steigender Frequenz immer kleiner werden.

Ist zum Beispiel bei der Grund-Periodenlänge 568 (110 Hz = A") ein Ausschlag von 8 Einheiten gerade wahrnehmbar (Abstand zum nächsten Halbton ist hier etwa 33, also das Vierfache), so bewegt sich dasselbe "Vibrato" drei Oktaven höher bei 880 Hz = a' bereits über +/– zwei Halbtöne! Die Periodenlänge dieses a' beträgt 71 und der Abstand zum nächsten Halbton nur noch etwa 4 Einheiten (vergleiche die Tabelle im Anhang).

Ähnlich wie bei der Amplitude kann man auch bei Tonperioden-Hüllkurven in einem Abschnitt die Tonperiodenlänge auf einen absoluten Wert neu festsetzen. Diese Möglichkeit ist im BASIC-Handbuch weder erwähnt noch beschrieben, obwohl es auch hier möglich ist! Die Syntax ist dabei der Definition einer Hardware-Volumen-Hüllkurve ähnlich:

```
ENT 5,  =568,10, ...
```

Auch hier enthält der Abschnitt nur noch zwei statt drei Parameter. Der erste gibt die neue Tonperiodenlänge an, und der zweite bestimmt die Pausenlänge nach dieser Änderung. Da auch für jedes ENT-Treppchen zuerst die Änderung durchgeführt und erst danach gewartet wird, kann man, wenn man bereits im ersten Abschnitt eine absolute Periode programmiert, die Angabe im SOUND-Statement vollkommen unterdrücken.

In Maschinensprache sieht die Syntax ein wenig anders aus. Hier ist die absolute Festlegung daran erkennbar, daß im ersten Byte des Abschnitts (normalerweise Schrittzahl) die obersten 4 Bits gesetzt sind. Damit ergibt sich für dieses Byte ein Wert, der größer oder gleich &F0 = 240 ist. Alle Werte darunter (0 bis &EF = 239) zeigen, daß ein relativer Abschnitt folgt. Alle Werte ab &F0 zeigen, daß eine absolute Festlegung der Frequenz erfolgen soll.

Hierbei werden dann die ersten beiden Bytes zu einem Word zusammengefaßt, das die neue Periodenlänge darstellt. Im ersten Byte werden die vier obersten Bits ausmaskiert, und der PSG wird mit dem so erhaltenen Wert programmiert. Da die Periodenlänge beim AY-3-8912 nur mit 12 Bits genau angegeben werden kann, sind die 4 obersten Bits bedeutungslos.

Das dritte Byte stellt dann wieder, wie gewöhnlich, die Pausenlänge dar, die nach dieser Einstellung abgewartet werden muß.

Vor einer gemeinen Falle seien hier aber die Assemblerprogrammierer gewarnt: Das erste Byte der Periodenlänge ist hier das höherwertige Byte! Die beiden folgenden Beispiele zeigen einen falschen und einen korrekten Abschnitt für den Parameterblock:

Einstellung einer absoluten Periodenlänge (APL) in Assembler:

falsch:
```
DEFW &F000 + APL
DEFB PAUSE
```

richtig:
```
DEFB APL\256 OR &F0 ; MSB von APL
DEFB APL AND &FF    ; LSB von APL
DEFB PAUSE
```

Viele Assembler stellen für die benötigten Verknüpfungen spezielle Kommandos bereit. Leider gibt es hier keinen einheitlichen Standard. Für das Beispiel wurde deshalb die Syntax von BASIC angenommen.

Bedienen bei Bedarf

Wenn ein Programm nicht gerade den Zweck hat, ein Musikstück oder Geräusche wiederzugeben, wird man die Sound-Programmierung mit möglichst wenig verbrauchter Zeit nebenher erledigen wollen.

Der Sound-Manager verfügt pro Kanal über eine Warteschlange für vier Töne, wobei der erste normalerweise gerade abgespielt wird. Man kann also zunächst einmal 4 Töne an den Sound-Manager übergeben, ohne mit einer Verzögerung rechnen zu müssen. Schiebt man dann aber auch gleich den fünften nach (oder versucht man das zumindest), so muß das Programm warten, bis wieder ein Platz in der Warteschlange frei wird.

Bei ungeschickter Programmierung wird die CPU die meiste Zeit warten, um mit dem nächsten Sound-Befehl fortzufahren.

Die einfachste Möglichkeit (für die Programmierer des Sound Managers) besteht zunächst einmal darin, mit der Funktion SQ(kanal) (Vektor &BCAD SOUND CHECK in Maschinensprache) regelmäßig nachzufragen, ob in einem Kanal wieder ein Platz frei ist, um nur dann den nächsten Sound-Befehl nach-

zuschieben. Der Wert, den man dabei erhält, ist Bit-signifikant und liefert folgende Informationen:

Kanalstatus:

Bits 0, 1, 2 – Anzahl freier Plätze
Bits 3, 4, 5 – Ist eins dieser Bits gesetzt, so wartet der erste Ton des getesteten Kanals auf ein Rendezvous mit einem anderen. Dabei sind die einzelnen Kanäle in ihrer üblichen Reihenfolge codiert: 3, 4, 5 entsprechen A, B, C.
Bit 6 = 1 – Der erste Ton in der Warteschlange befindet sich im Haltezustand (Hold-Bit gesetzt).
Bit 7 = 1 – Der erste Ton der Schlange wird gerade gespielt. Bit 6 und 7 können nicht gleichzeitig gesetzt sein. Ebenso ist bei einem gesetzten Bit 7 die Information in den Bits 3 bis 5 bedeutungslos.

Der Nachteil bei dieser Methode, dem sogenannten Polling ist, daß man an jeder nur denkbaren Stelle im Hauptprogramm einen solchen Test einfügen muß. Dadurch wird das Programm nicht nur länger, sondern auch langsamer, und die nahtlose Versorgung des Sound-Managers ist trotzdem nicht immer sichergestellt.

Deshalb ist es auch möglich, den Sound-Manager via Software-Interrupt zu programmieren. Dabei kann man für jeden Kanal eine Routine bestimmen, die aufgerufen werden soll, wenn in seiner Warteschlange ein Platz frei wird.

In Maschinensprache muß man dem Vektor &BCB0 SOUND ARM EVENT einen Event-Block übergeben, den der Sound-Manager dann in seine SOUND CHAIN einreiht.

In BASIC dient dafür der Befehl ON SQ(kanal) GOSUB.

Dabei muß man beachten, daß diese Interrupts nur jeweils einmal ausgeführt werden und sich deshalb ständig selbst initialisieren müssen, *nachdem* ein neuer Sound-Befehl programmiert wurde. Der Interrupt für einen speziellen Kanal wird nämlich aufgehoben, wenn

1. der Interrupt erzeugt wurde;
2. ein Sound-Befehl in den entsprechenden Kanal nachgeschoben wurde oder
3. der Kanalstatus getestet wird (SOUND CHECK oder Funktion SQ(kanal)).

Da die Programmierung der drei Kanäle so vollkommen unabhängig voneinander erfolgt, kann ihre Synchronisierung mit der Zeit aus dem Tritt geraten. Am sinnvollsten ist es deshalb, alle Kanäle einmal pro Takt mit Hilfe der Rendezvous-Technik zu synchronisieren.

Sound und Interrupts in BASIC

Im Gegensatz zu den meisten anderen Firmware Packs ist die Behandlung von Interrupts in BASIC und in Maschinensprache doch recht unterschiedlich. Am leichtesten läßt sich noch die Programmierung des Sound-Managers per Interrupt von BASIC in Assemblerprogramme übersetzen.

Da gerade die Benutzung von Interrupt-Routinen in einem Programm besonders viel organisatorische Sorgfalt erfordert, ist es bestimmt nicht die schlechteste Idee, die Interrupt-Programmierung erst einmal in BASIC zu üben.

Der BASIC-Interpreter stellt insgesamt vier verschiedene Uhren zur Verfügung. Jede Uhr kann in Zeiteinheiten von 50stel-Sekunden programmiert werden. Dabei kann man entweder den Befehl AFTER verwenden, wodurch nach der eingestellten Zeit nur einmal eine Unterbrechung ausgelöst wird, oder man benutzt EVERY, dann wird die Unterbrechung fortlaufend im eingestellten Zeitintervall erzeugt.

Natürlich genügt es nicht, eine Unterbrechung nur "auszulösen". Diese Unterbrechung muß auch etwas bewirken. Deshalb muß man immer, wenn man eine Uhr programmiert, angeben, welche Programmzeile aufgerufen werden soll, wenn der Interrupt nun eintritt.

Dann wird das laufende Hauptprogramm unterbrochen und das angegebene Unterprogramm ausgeführt. Dieses kann abschließend ganz normal mit RETURN zum Hauptprogramm zurückkehren.

```
EVERY 50 GOSUB 2000
```
– ruft regelmäßig einmal pro Sekunde Zeile 2000 auf

```
AFTER 3000 GOSUB 5000
```
– ruft nach einer Minute die Zeile 5000 auf

Äußerst wichtig ist: Alle Interrupt-Unterprogramme müssen so gestaltet sein, daß sie sich nicht gegenseitig und auch nicht das Hauptprogramm beeinflussen.

Vor allem dürfen keine Variablen, die das unterbrochene Programm benutzen könnte, verändert werden. Auch Textausgaben dürfen bei dem unterbrochenen Programm nicht dazwischenkommen.

Sollen trotzdem Variablen oder Datenströme (u.a. eben die Textausgabe) von mehreren parallel laufenden Programmen benutzt werden, so muß man in den kritischen Bereichen Unterbrechungen verbieten: Dazu dienen die Befehle DI und EI.

```
100 EVERY 50 GOSUB 200
110 FOR I=1 TO 10000
120 DI:LOCATE 1,1:PRINT I*I:EI
130 NEXT:END
140 '
200 LOCATE 10,10:PRINT I:RETURN
```

In diesem Beispiel benutzen sowohl das Hauptprogramm als auch die Interrupt-Routine das Textfenster 0. Zwischen LOCATE und PRINT in Zeile 120 darf der Interrupt nicht auftreten, weil er dann die Position für die Textausgabe auf seinen eigenen Wert verstellen würde. Deswegen wird vorher der Interrupt verboten und erst nachher wieder zugelassen. Eine bessere Möglichkeit ist beim CPC aber sicher, für das Interrupt-Programm ein anderes Textfenster zu benutzen!

Andererseits ist es jedoch vollkommen gefahrlos, im Interrupt-Programm die Variable I auszudrucken, obwohl sie im Hauptprogramm verwendet wird. Sie wird ja nicht verändert.

Gibt man wie in diesem Beispiel keine Nummer für die zu verwendende Uhr an, so nimmt der BASIC-Interpreter immer Uhr 0 als Default.

Die vier Uhren unterscheiden sich in ihrer Priorität: Der Interrupt des einen Weckers kann den einer anderen Uhr nicht nach Belieben erneut unterbrechen. Nur die Uhren mit übergeordneten Prioritäten können das. Dabei haben die Uhren in der Reihenfolge ihrer Numerierung aufsteigende Dringlichkeit: Der Timer 3 ist am wichtigsten und kann alle anderen unterbrechen, selbst aber von keinem anderen Interrupt unterbrochen werden.

Die Interrupt-Möglichkeit des Sound-Managers ist ebenfalls genutzt. Hier werden die Unterbrechungen aber nicht in regelmäßigen Abständen erzeugt, sondern, für jeden Kanal getrennt programmierbar, sobald in der Ton-Warteschlange eines Kanals ein Platz frei wird. Alle drei Kanäle haben dabei die gleiche Priorität wie der Timer 2. Sound-Interrupts und der Timer 2 können sich also nicht gegenseitig unterbrechen! Die Syntax ist:

`ON SQ(kanal) GOSUB zeile` (kanal ist Bit-signifikant: 1=A / 2=B / 4=C)

Von absolut übergeordneter Priorität ist aber das Break-Event, das sich auch mit DI nicht ausschalten läßt. Speziell hier scheint leider im BASIC-Interpreter oder im Kernel des CPC 464 ein Fehler vorzuliegen. Der Befehl ON_BREAK_GOSUB bereitet ständig Schwierigkeiten, wenn parallel noch weitere Interrupts laufen. Mit schönster Unregelmäßigkeit schaltet sich dieser Interrupt selbst und/oder auch andere Unterbrechungsanweisungen aus.

Eine BASIC-Demo

So werden im nachstehenden Demo-Programm ständig die Sound-Interrupts abgeschaltet. Deshalb sollten CPC 464-Benutzer in Zeile 1510 für das Break-Unterprogramm nur ein RUN eintragen.

Das hat zwar zur Folge, daß danach erst einmal 8 Sekunden lang die Halbtonschritte berechnet und die DATA-Zeilen wieder ausgelesen werden, das ist aber immer noch besser, als plötzlich überhaupt kein Ton mehr.

Das Programm nutzt alle Interrupt-Timer aus, und auch die Begleitmusik wird auf allen drei Tonkanälen via Interrupt nachgeschoben. Mit Hilfe des Break-Events, das leider beim CPC 464 mit kleinen Fehlern behaftet ist, kann jederzeit das Programm neu gestartet werden.

Animation durch Umfärben der Tinten

Die Grafik, die das Programm ausgibt, wird über die Farbzuordnung (INK-Statement) bewegt: Der Linienzug, der durch das Hauptprogramm in den Zeilen 1610 bis 1660 gezeichnet wird, durchläuft regelmäßig die Tintennummern 1 bis 14. Das Interrupt-Programm in Zeile 1790 programmiert diese Tinten nun zyklisch zwischen Farbe 1 (blau, wie der Hintergrund) und der Farbe fa um. Dadurch scheint durch den Linienzug eine Lücke durchzulaufen, was eine Bewegung vortäuscht. Zusätzlich verändert das Interrupt-Programm in Zeile 1920 die Variable fa. Dadurch werden die Tinten immer wieder in neuen Farben gesetzt.

Tonprogrammierung

Während das Hauptprogramm läuft und Interrupt-gesteuert animiert wird, wird parallel dazu, ebenfalls per Interrupt, ein Musikstück abgespielt.

Bei der Initialisierung werden zunächst zwei Volumen-Hüllkurven definiert (Zeilen 390, 400) und dann die Periodenlängen für die Halbtöne von insgesamt 7 Oktaven berechnet. In Zeile 600 wird eine Funktion definiert, die zusammen mit dem Feld ht(tonart,note) aus einer gegebenen Tonart, Oktave und Ganztonnummer den zugehörigen Halbton berechnet.

Das wird im darauffolgenden Kanon benötigt. Scott E. Kim kanonisierte nämlich in "Good King Wenceslas" ein Thema, indem er die erste Stimme in der zweiten mit einem halben Takt Versatz und auf dem Kopf wiederholte! "Auf dem Kopf" bezieht sich dabei nicht auf Halbtonschritte, sondern eben auf ganze.

In den DATA-Zeilen ab Zeile 780 sind jeweils paarweise übereinander die Ganzton-Nummer (relativ zu einer festen Note, in diesem Fall Mittel-C) und die Notenlänge (in Viertelnoten) abgelegt. Bei den meisten Liedern wird man die Halbtonnummern speichern und auch eventuell für die Notenlängen ein feineres Raster anlegen müssen. Letzteres richtet sich hauptsächlich nach der kürzesten Note, die im gesamten Musikstück vorkommt.

Da S.E. Kim aber nur ganze Noten verwendet, also keine Kreuze oder b-Noten benutzt, kann man diese praktischerweise auch gleich in die DATA-Zeilen eintragen. Und da sich die Inversion der Notenhöhe in der zweiten Stimme auf Ganztöne bezieht, kann man sich die DATA-Zeilen für die zweite Stimme sparen.

Nachdem die DATA-Zeilen durch das Unterprogramm ab Zeile 1400 in die beiden Zahlenfelder l(x) für die Notenlänge und t(x) für die Notenhöhe ausgelesen wurden, um einen wahlfreien Zugriff auf die einzelnen Noten zu haben, werden die Interrupts für die drei Kanäle A (Oberstimme), C (zweite Stimme) und B (Percussion) angestoßen.

Wenn man sich die Interrupt-Routinen für die Stimmen A und C genau ansieht, so erkennt man, daß auch die zweite Stimme die Noteninformationen der ersten Stimme benutzt: Während Kanal A die Ganztonnote als t(t1) bestimmt, berechnet Kanal C sie als 7-t(t2). t1 und t2 sind dabei die Zeiger der beiden Kanäle in den Notenfeldern.

Synchronisiert werden die drei Kanäle in zwei Grüppchen: Zu jedem Taktanfang hat Kanal A ein Rendezvous mit der Percussion und in der Taktmitte mit der Gegenstimme. Damit wird übrigens auch gleich der Versatz um einen halben Takt zwischen Kanal A und C erreicht: Während Kanal A in der Taktmitte sein erstes Rendezvous mit C hat, hat dieser Kanal (scheinbar) das Rendezvous zu seinem Taktanfang. Kanal C kann also erst loslegen, wenn Kanal A schon bis zur Mitte des zweiten Taktes gekommen ist.

```
100 ' ************************************************************
110 ' **                                                        **
120 ' **    Interrupt & Sound & Grafik   -   Demo      vs. 29.5.86 **
130 ' **                                                        **
140 ' ************************************************************
150 '
160 KEY DEF 66,0,140,140,140:KEY 140,CHR$(&EF)+CHR$(252)     ' Break-Taste
170 DEFINT a-z
180 BORDER 0:INK 0,0:MODE 0:PAPER 0:PEN#1,15:PEN#2,15:PEN#3,15:INK 15,26
190 ORIGIN 320,200
200 WINDOW 2,19,2,24
210 DEF FNz$(z)=CHR$(48+z\10)+CHR$(48+z MOD 10)
220 '
230 GOSUB 1920                                ' After#0 --> Farbenwechsel
```

```
                                          ' starten
240 t=INT(TIME/300):st=t\3600:t=t MOD 3600  '    Zeit seit letztem Reset
250 mn=t\60:sk=t MOD 60                     '    berechnen
260 EVERY 50,2 GOSUB 1850                  ' Every#2 --> mitlaufende Uhr
270 s=1:EVERY 5,3 GOSUB 1790               ' Every#3 --> INKs zyklisch
                                           ' faerben
280 ON BREAK GOSUB 1510                    ' Break-Event
290 xx=1:yy=1:EVERY 7,1 GOSUB 1980         ' Every#1 --> Sternchengimmik
300 '
310 GOSUB 390       ' Allgemeine Sound-Initialisierungen
320 GOSUB 870       ' initialisiere:  Good King Wenceslas
330 GOTO 1580       ' Grafik-Demo
340 '
350 ' +-----------------------------------------------------+
360 ' !   **********   SOUND-Initialisierung   **********   !
370 ' +-----------------------------------------------------+
380 '
390 ENV 1,  4,2,1,  2,-1,2,  1,0,50, 5,-1,15 ' Amplituden-Huellkurven
                                           ' fuer
400 ENV 2,  3,-1,2, 3,-1,20                ' Musikstimmen und
                                           ' Percussion
410 '
420 i=7*12+1:DIM plen(i)                ' 7 Oktaven mit je 12 Toenen
430 pause=i:plen(i)=0                   ' Pause = kein Ton ->
                                        ' Periodenlaenge = 0
440 kton.A!=1/440 * 1000000/16          ' Kammerton A (Ton 9 in Oktave 3)
450 '
460 ' +-----------------------------------------------------+
470 ' !   **********   Halbtoene einer Oktave:   *********   !
480 ' !   C  CIS  D  DIS  E  F  FIS  G  GIS  A  AIS  H  C    !
490 ' +-----------------------------------------------------+
500 '
510 FOR halbton = 0 TO 7*12                      ' Berechnung aller
520   plen(halbton)=kton.A!*2^(3+(9-halbton)/12) ' Periodenlaengen
530 NEXT                                         ' in sieben Oktaven
540 '
550 ' +-----------------------------------------------------+
560 ' ! *** Funktion zur Berechnung des Halbtonschrittes *** !
570 ' ! ***   Halbton = FNhalbton (tonart,oktave,note)   *** !
580 ' +-----------------------------------------------------+
590 '
600 DEF FNhalbton(t,o,n)=(o+INT(n/7))*12+ht(t,(n+70)MOD 7)
610 '
620 DIM ht(1,7):RESTORE 710                      ' ht() enthaelt die
630 dur=0:moll=1                                 ' Halbtonnummern fuer
640 FOR tonart=dur TO moll                       ' die ganzen Noten
650   FOR note=0 TO 7                            ' in C-Dur und C-Moll
660     READ ht(tonart,note)
670   NEXT
680 NEXT
690 RETURN
700 '          <----- Dur ------>    <----- Moll ---->
710 DATA       0,2,4,5,7,9,11,12,    0,2,3,5,7,8,10,12
720 '
730 ' +-----------------------------------------------------+
```

```
740 ' ! ******** Kanon durch Umkehrung der Tonhoehe: ******* !
750 ' ! ******** Good King Wenceslas (Scott E. Kim) ******** !
760 ' +-----------------------------------------------------+
770 '
780 DATA 0,0,0,1,  0,0,-3,  -2,-3,-2,-1,  0,0,  0,0,0,1,  0,0,-3,  99
790 DATA 1,1,1,1,  1,1, 2,   1, 1, 1, 1,  2,2,  1,1,1,1,  1,1, 2,  99
800 '
810 DATA -2,-3,-2,-1,  0,0,  4,3,2,1,  2,1,0,  -2,-3,-2,-1,  0,0, 99
820 DATA  1, 1, 1, 1,  2,2,  1,1,1,1,  1,1,2,   1, 1, 1, 1,  2,2, 99
830 '
840 DATA -3,-3,-2,-1,  0,0,1,  4,3,2,1,  0,0,   99
850 DATA  1, 1, 1, 1,  1,1,2,  1,1,1,1,  2,2,   100
860 '
870 RESTORE 780:GOSUB 1400                    ' Auslesen der DATA-Zeilen
880 tempo=45:oktave=3:tonart=dur               ' Einstellungen
890 t1=1:l1=0:GOSUB 980                        ' Init Oberstimme:  Kanal A
900 t2=1:l2=2:GOSUB 1100                       ' Init Gegenstimme: Kanal B
910 t3=0     :GOSUB 1210                       ' Init Percussion: Kanal C
920 RETURN
930 '
940 ' +-----------------------------------------------------+
950 ' !   ***********   Kanal A (Oberstimme)   ***********   !
960 ' +-----------------------------------------------------+
970 '
980  status=1:IF l1=2 THEN status=33          ' Rendezvous mit Gegenstimme
990           IF l1=0 THEN status=17          ' Rendezvous mit Percussion
1000 laenge=l(t1):note=t(t1):t1=t1 MOD pw +1
1010 l1=(l1+laenge)MOD 4:laut=5
1020 GOSUB 1320                               ' Tonausgabe
1030 ON SQ(1) GOSUB 980                       ' Re-Init
1040 RETURN
1050 '
1060 ' +-----------------------------------------------------+
1070 ' !   **********   Kanal C (Gegenstimme)   ***********   !
1080 ' +-----------------------------------------------------+
1090 '
1100 status=4:IF l2=2 THEN status=12          ' Rendezvous mit Oberstimme
1110 laenge=l(t2):note=-7-t(t2):t2=t2 MOD pw +1
1120 l2=(l2+laenge)MOD 4:laut=7
1130 GOSUB 1320                               ' Tonausgabe
1140 ON SQ(4) GOSUB 1100                      ' Re-Init
1150 RETURN
1160 '
1170 ' +-----------------------------------------------------+
1180 ' !   ***********   Kanal B (Percussion)   ***********   !
1190 ' +-----------------------------------------------------+
1200 '
1210 status=2:IF t3=2 THEN status=10          ' Rendezvous mit Oberstimme
1220 noise=(2 - t3 MOD 2)*10                  ' Rauschhoehe
1230 SOUND status,0,tempo,13,2,0,noise
1240 t3=(t3+1) MOD 4
1250 ON SQ(2) GOSUB 1210                      ' Re-Init
1260 RETURN
1270 '
1280 ' +-----------------------------------------------------+
```

```
1290 ' !   ***********    Ausgabe einer Note    ***********   !
1300 ' +-------------------------------------------------------+
1310 '
1320 halbton=FNhalbton(tonart,oktave,note)
1330 SOUND status,plen(halbton),tempo*laenge,laut,1
1340 RETURN
1350 '
1360 ' +-------------------------------------------------------+
1370 ' !  Auslesen der Tonhoehen und -laengen aus DATA-Zeilen  !
1380 ' +-------------------------------------------------------+
1390 '
1400 DIM t(100),l(100)
1410 t=1:l=1
1420 READ t(t):IF t(t)<99 THEN t=t+1:GOTO 1420 ' Tonhoehen
1430 READ l(l):IF l(l)<99 THEN l=l+1:GOTO 1430 ' Notenlaengen
1440 IF l(l)=99 THEN 1420
1450 pw=t-1:RETURN                              ' Laenge der
                                                ' Wiederholungsperiode
1460 '
1470 ' +-------------------------------------------------------+
1480 ' !  ******  Unterprogramm bei Break-Erkennung  *******  !
1490 ' +-------------------------------------------------------+
1500 '
1510 RUN           ' CPC 464: Neustart. Anders geht's nicht. Sonst gehen
1520 '             ' irgendwie programmierte Interrupts verloren.
1510 brk=1:RETURN  ' Beim CPC 664 und 6128 gibt's keine Probleme. Die
1520               ' Musik-Ausgabe muss nicht erneut initialisiert
                   ' werden.
1530 '
1540 ' +-------------------------------------------------------+
1550 ' ! *******    Hauptprogramm: Grafik-Animation *******   !
1560 ' +-------------------------------------------------------+
1570 '
1580 CLS:i1=5+RND*20:i2=5+RND*20        ' Frequenzverhaeltnis x zu y
1590 f!=i1/i2:zfa=1:brk=0
1600 '
1610 DEF FNx(i!)=SIN(i!)*285            ' X-Y-Ueberlagerung
1620 DEF FNy(i!)=COS(i!*f!)*180         ' zweier Sinus-Schwingungen
1630 PLOT FNx(0),FNy(0)
1640 FOR i!=0 TO i2*2*PI STEP 2*PI/INT(3+20*RND)/14
1650   DRAW FNx(i!),FNy(i!),zfa:zfa=zfa MOD 14+1:IF brk THEN 1580
1660 NEXT
1670 '                                  ' abschliessend Laufschrift:
1680 '
1690 t$=" *** reines Basic-Demo *** 100% Maschinencode-frei"
1700 t$=SPACE$(16)+t$+t$+t$+t$+" ***"+SPACE$(16)
1710 FOR i=1 TO LEN(t$)-16
1720   LOCATE#3,3,12:PRINT #3,MID$(t$,i,16):IF brk THEN 1580
1730 NEXT: GOTO 1580
1740 '
1750 ' +-------------------------------------------------------+
1760 ' ! Interrupt-Programm: Farbe fa durch die Inks schieben !
1770 ' +-------------------------------------------------------+
1780 '
1790 INK s,fa:s=s MOD 14+1:INK (s+2)MOD 14+1,0:RETURN
```

```
1800 '
1810 ' +-----------------------------------------------------+
1820 ' !  ****** Interrupt-Programm: Uhrzeit ausgeben ******  !
1830 ' +-----------------------------------------------------+
1840 '
1850 sk=sk+1:IF sk=60 THEN sk=0:mn=mn+1:IF mn=60 THEN mn=0:st=(st+1)MOD
24
1860 LOCATE#1,7,1:PRINT#1,FNz$(st);":";FNz$(mn);":";FNz$(sk):RETURN
1870 '
1880 ' +-----------------------------------------------------+
1890 ' !  *******  Interrupt-Programm: Farbe aendern  ******* !
1900 ' +-----------------------------------------------------+
1910 '
1920 fa=7+INT(RND*20):AFTER 10+RND*20,0 GOSUB 1920:RETURN
1930 '
1940 ' +-----------------------------------------------------+
1950 ' ! *** Interrupt-Programm: Sternchengimmik am Rande *** !
1960 ' +-----------------------------------------------------+
1970 '
1980 LOCATE#2,xx,yy:PRINT#2," ";
1990 LOCATE#2,21-xx,26-yy:PRINT#2," ";
2000 IF xx<20 THEN xx=xx+1 ELSE yy=yy+1:IF yy=26 THEN xx=1:yy=1
2010 LOCATE#2,xx,yy:PRINT#2,"*";
2020 LOCATE#2,21-xx,26-yy:PRINT#2,"*";
2030 RETURN
```

DER KERNEL – SOFTWARE-INTERRUPTS

Unter dem Kernel (Kern) eines Betriebssystems versteht man immer die Schaltzentrale, den Manager des Ganzen. Im Schneider CPC umfaßt der Kernel alle Routinen, die sich mit der Speicherverwaltung beschäftigen, dem Interrupt-Mechanismus und der Behandlung externer Kommandos.

Die Speicherverwaltung wird über den LOW KERNEL JUMPBLOCK und den HIGH KERNEL JUMPBLOCK gesteuert und wurde im Kapitel über die Speicheraufteilung des CPC und in den darauf folgenden dargestellt, ebenso die Behandlung der RSX-Kommandos und Hintergrund-ROMs.

Was noch bleibt, ist der Software-Interrupt-Mechanismus.

Obwohl der Schneider CPC nur eine einzige Interrupt-Quelle kennt (sofern er nicht um Interrupt-erzeugende Module erweitert worden ist), können praktisch beliebig viele Software-Interrupts simuliert werden.

Die Quelle fuer den Hardware-Interrupt ist die ULA, die 300mal pro Sekunde, teilweise synchronisiert mit dem Strahlhochlauf des Monitorbildes, ein Interrupt-Signal für die CPU ausgibt. Der Kernel erzeugt daraus alle weiteren Software-Unterbrechungen. Diese werden auch "Ereignis" oder, "Event" genannt.

Das Prinzip

Um den Event-Mechanismus überhaupt zu durchschauen, muß man sich unbedingt dessen grundsätzlichen Aufbau klarmachen:

Alle Software-Interrupts werden über Datenblöcke gesteuert, die der Kernel in eine seiner vielen Listen einreiht. Der elementarste Datenblock ist dabei der sogenannte "Event-Block". Für eine vollständige Interrupt-Behandlung muß man zunächst einmal:

1. ein Event programmieren

Dieses Ereignis wird dann, je nach Programmierung, vom Kernel (oder einer anderen Routine) angestoßen:

2. Das Event wird "gekickt"

Damit ist das Event (oder der Event-Block, je nach Sichtweise) zwar aktiviert, die zugehörige Interrupt-Routine aber noch nicht behandelt. Das ist erst der dritte Schritt, der mitunter mit beträchtlicher Verzögerung getrennt vollzogen wird:

3. Die Event-Behandlungsroutine wird aufgerufen

Interrupt-Quellen

Insgesamt gibt es bisher 6 verschiedene mögliche "Quellen", mit deren Hilfe Ereignisse programmiert werden können. Das sind zunächst einmal drei verkettete Listen, in die man Datenblocks eintragen kann (via Vektoren im MAIN FIRMWARE JUMPBLOCK):

1. die FAST TICKER CHAIN
2. die FRAME FLYBACK CHAIN
3. die TICKER CHAIN

4. Darüber hinaus unterhält der Kernel eine weitere Liste, die aber ausschließlich dem SOUND MANAGER zur Verfügung steht: Der Sound Manager bietet aber die Möglichkeit, für jeden der drei Tonkänale ein Event für den Fall zu programmieren, daß ein Platz in seiner Ton-Warteschlange frei wird.

5. Auch den Key-Manager kann man veranlassen, ein Break-Event zu erzeugen, wenn der Anwender auf die ESC-Taste drückt.

6. Und zuletzt besteht noch die Möglichkeit, über eine bestimmte Routine des Kernel ein Event direkt zu kicken. Das ist zwar nach obiger Einteilung bereits die zweite Phase, eben der "Kick". Hiermit kann man sich aber seine eigenen Interrupt-Quellen basteln:

Die Behandlungsroutine eines externen Interrupts kann hiermit ein Event antreten. Dadurch wandelt man praktisch den Hardware-Interrupt in sein Software-Äquivalent um. Oder man kann sich eigene Listen stricken, die beispielsweise nur einmal pro Sekunde oder bei speziellen Bedingungen (bei jedem Tastendruck, Feuer auf dem Joystick o.ä.) abgearbeitet werden.

Das Anstoßen der einzelnen Event-Blocks wird dann vom Kernel besorgt oder, über den Vektor &BCF2 KL EVENT, von jeder anderen Einrichtung, die sich dazu berufen fühlt (beispielsweise von einem externen Interrupt).

Die Chains des Kernel

Der Kernel stellt für den Anwender direkt drei Listen bereit, in die man Datenblöcke eintragen kann, um so Software-Interrupts zu programmieren. Diese Listen unterscheiden sich hauptsächlich darin, wann sie vom Kernel berücksichtigt werden.

Die FAST TICKER CHAIN wird mit jedem Interrupt, den die ULA erzeugt, überprüft. Hiermit können Events programmiert werden, die 300mal pro Sekunde behandelt werden müssen. Der Fast Ticker ist zwar der schnellste, verbraucht deshalb aber auch die meiste Rechenzeit. Drei oder vier längere Routinen, auf dem Fast Ticker eingehängt, können das Hauptprogramm mitunter ganz zum Stillstand bringen.

Zum Einhängen eines Datenblocks in der Fast Ticker Chain kann man den Vektor &BCE3 KL ADD FAST TICKER benutzen. In HL muß man dabei einen Zeiger auf den Datenblock, den "Fast Ticker Block" übergeben. Dieser ist wie folgt aufgebaut:

Aufbau des Fast Ticker Blocks:

Byte 0, 1	–	Platz für den Hangel-Pointer in der Fast Ticker Chain
Byte 2 ff.	–	Der Event-Block

Eine weitere Liste ist die FRAME FLYBACK CHAIN, die mit jedem Strahlhochlauf auf dem Monitor dran ist. Das ist, je nach Verkaufsland, 50 oder 60mal in der Sekunde der Fall. In Europa, wo man überall mit einer Netzfre-

quenz von 50 Hz arbeitet, schreiben die verschiedenen Fernsehnormen (z.B. PAL oder SECAM) auch eine Bildfrequenz von 50 Hz vor. In Amerika arbeitet man mit der NTSC-Norm und 60 Hertz.

Diese Interrupt-Quelle ist für alle Aktionen auf dem Bildschirm gedacht, die, würden sie sichtbar, ein Flackern oder andere Störungen verursachen würden.

So ist in dieser Liste standardmäßig das Farbenblinken des Screen Packs eingehängt. Neu programmierte INK-Farbzuordnungen werden ebenfalls erst mit dem nächsten Strahlhochlauf in die ULA geschrieben.

Speziell in Spielen mit schnell bewegter Sprite- (oder Shape-) Grafik werden die Figuren immer während der Austastlücke bewegt. Ein gängiger Algorithmus für Sprites ist:

Sprite plazieren: Grafikinformation des dadurch verdeckten Bildausschnittes retten, dann Sprite in den Bildschirm malen.

Sprite entfernen: Gerettete Grafikinformation in den Bildschirm zurückkopieren.

Sprite bewegen: Sprite entfernen. Koordinaten verschieben. Sprite plazieren.

Um ein Sprite zu bewegen, wird kurzzeitig der verdeckte Hintergrund im Video-RAM restauriert. Der soll natürlich nicht zu sehen sein, weil sonst die Figur flimmern würde. Deshalb werden diese Manipulationen möglichst vorgenommen, wenn kein Bild dargestellt wird, was in der Austastlücke der Fall ist.

Um einen Block einzuhängen, kann man den Vektor &BCDA KL ADD FRAME FLY benutzen. Der "Frame Flyback Block" ist dabei genauso aufgebaut wie der "Fast Ticker Block:

Aufbau des Frame Flyback Blocks:

Byte 0, 1	– Platz für den Hangel-Pointer in der Frame Flyback Chain
Byte 2 ff.	– Der Event-Block

Am komfortabelsten ist die TICKER CHAIN, die allerdings nur 50mal pro Sekunde abgearbeitet wird. Während in den anderen Listen ohne zusätzlichen Aufwand nur solche Ereignisse programmiert werden können, die immer wieder "angestoßen" werden (Repeater), kann man bei der TICKER CHAIN auch sogenannte "one shots", also einmalige Ereignisse, definieren.

BASIC benutzt für EVERY und AFTER ausschließlich die TICKER CHAIN, was man auch daran erkennen kann, daß die Zeiten hier in 50stel Sekunden angegeben werden müssen.

Über den Vektor &BCE9 KL ADD TICKER kann man einen Ticker Block in die Ticker Chain einhängen. Dieser Block ist etwas anders aufgebaut als die beiden anderen. Beim Aufruf des Vektors muß man im DE- und BC-Register die gewünschten Werte für den Count Down (Startverzögerung) und Reload Count (Nachladewert für Repeater) übergeben:

Aufbau eines Ticker Blocks:

Byte 0, 1	– Platz für den Hangel-Pointer in der Ticker Chain
Byte 2, 3	– Count Down = Zähler für die Ticker-Zahl bis zum ersten Interrupt
Byte 4, 5	– Reload Count = Nachladewert für weitere Interrupts; bei &0000 wird nur ein Interrupt erzeugt
Byte 6 ff.	– Der Event-Block

Events

Nachdem nun die Interrupt-Quellen des Kernel beschrieben worden sind, kommen wir zum Interrupt selbst. Dafür muß man den Event-Block betrachten. Dieser Standard-Datenblock ist für die "Reaktionsseite" zuständig. Ist ein Event-Block erst einmal angestoßen (gekickt), so sieht man ihm nicht mehr an, woher der Anstoß kam. Jeder Event-Block ist wie folgt aufgebaut (*Achtung:* Die Angaben im Firmware Manual hierzu stimmen nicht!):

Der Event-Block:

Byte 0, 1	–	Platz für den Hangel-Pointer in einer "Pending Queue"
Byte 2	– COUNT	= Kick-Zähler und Steuerbyte
Byte 3	– CLASS	= Interrupt-Typ
Byte 4, 5	– ADDRESS	= Adresse der Behandlungsroutine
Byte 6	– ROM	= nur bei Bedarf: benötigte ROM-Konfiguration
Byte 7ff	–	nur bei Bedarf: lokale Variablen für die Behandlungsroutine

Wird ein solcher Event-Block gekickt, so reiht ihn der Kernel entsprechend seines Typs in eine von zwei möglichen "abhängigen" Ketten, den sogenannten "Pending Queues" ein (wenn er nicht bereits drin ist). Dafür müssen die beiden ersten Bytes bereitgestellt werden. (Davon gibt es zwei Ausnahmen, die weiter unten beschrieben sind.)

Gleichzeitig wird der Kick-Zähler erhöht. Dadurch können kurzzeitig mehr Interrupt-Anforderungen eingehen, als die CPU abarbeiten kann. Die Anzahl der noch ausstehenden Kicks wird in diesem Byte gespeichert. Dieser Zähler geht aber nur bis +127. Negative Werte (der Kernel selbst benutzt immer -64 = &C0) bedeuten, daß dieser Event-Block ruhiggestellt ist und nicht mehr angestoßen werden soll. Dieses Byte ist also nicht nur ein Zähler, es hat auch Steuerfunktionen.

In die Bytes 4 bis 6 muß die Adresse der Behandlungsroutine für den Software-Interrupt eingetragen werden. Befindet sich diese Routine im zentralen RAM, so kann man auf die Angabe der ROM-Konfiguration verzichten. Liegt sie aber in einem ROM, wie das beim BASIC-Interpreter der Fall ist, so muß man auch noch das letzte Byte angeben. Diese drei Bytes bilden dann zusammen die FAR ADDRESS für einen Restart 3 (LOW KL FAR CALL).

Byte 3 ist der Interrupt-Typ. Hiermit werden drei verschiedene Daten des Event-Blocks beeinflußt: Bit 0 gibt an, ob die Adresse der Behandlungsroutine mit oder ohne Angabe der ROM-Konfiguration erfolgt, Bit 7 unterscheidet "synchrone" und "asynchrone" Events. Bit 6 unterscheidet extrem dringende ("express") und "normale" Unterbrechungen, und die restlichen Bits bestimmen bei einem synchronen Event dessen Dringlichkeit, die Priorität:

Das Typ-Byte im Event-Block:

Bit 0	= 1	– "near address" (keine ROM-Konfiguration angegeben)
	= 0	– "far address" (ROM-Konfiguration wie für RST 3 angegeben)
Bit 1–4	=	– Priorität (bei synchronen Events)
Bit 5	=	– muß auf 0 gesetzt werden
Bit 6	= 0	– normales Event
	= 1	– express Event
Bit 7	= 0	– synchrones Event
	= 1	– asynchrones Event

Express-asynchrone Events

"Express Asynchronous Events" (Bit 6 und Bit 7 gesetzt) sind Interrupts der Dringlichkeitsstufe 1: Nur bei solchen Ereignissen wird die zugehörige Behandlungsroutine *sofort* aufgerufen, sobald der Kernel ihr Eintreffen erkannt hat, und nicht erst in die "Asynchronous Pending Queue" eingereiht.

Nur diese Events werden noch bei ausgeschaltetem Hardware-Interrupt bearbeitet. Das hat mehrere Konsequenzen.

So sollte die Behandlungsroutine so kurz wie überhaupt nur möglich sein. Restarts etc. dürfen nicht benutzt werden, weil die allesamt Interrupts wieder zulassen würden.

Deshalb darf man für express-asynchrone Events auch keine "far address" angeben: Die würde ja mittels Restart 3 aufgerufen! Hier darf man nur eine "near address" ohne Angabe der ROM-Konfiguration benutzen. Damit muß die Behandlungsroutine gezwungenermaßen im zentralen RAM (oder Betriebssytem-ROM) liegen.

Allgemein gilt für asynchrone Events, daß sie die meisten Routinen des Betriebssystems nicht aufrufen dürfen, weil diese nicht reentrant (vielfach aufrufbar) sind.

Für die Behandlungsroutinen aller Events gelten die folgen Ein- und Aussprungsbedingungen (*Achtung:* Die Angaben hierzu im Firmware Manual stimmen nicht!):

Ein/Ausgabebedingungen für jede Event-Behandlungsroutinen:

1. near address:	Eingaben:	DE zeigt auf Byte 5 des Event-Blocks. Evtl. benötigte lokale Variablen für die Routine liegen ab Byte 7, also ab DE+2.
	Ausgaben:	IX und IY dürfen nicht verändert werden.
2. far address:	Eingaben:	HL zeigt auf Byte 4 des Event-Blocks. Evtl. benötigte lokale Variablen liegen also ab HL+3. Bei Hintergrund-ROMs zeigt IY auf die Unterkante des über HIMEM für das ROM reservierten Speicherbereichs.
	Ausgaben:	IX und IY dürfen nicht verändert werden.

Normal-asynchrone Events

Alle anderen, die "normalen" asynchronen Events reiht der Kernel zunächst einmal in die "Asynchronous Pending Queue" ein. Ist die Behandlung des Hardware-Interrupts beendet, so läßt der Kernel Interrupts wieder zu und ruft erst jetzt alle Einträge in dieser Warteliste auf, bevor er dann endgültig zum Hauptprogramm zurückkehrt.

Weil nun Interrupts wieder zugelassen und auch die normalen Registerverhältnisse in der CPU wieder hergestellt sind, können die Behandlungsroutinen für asynchrone normale Events zumindest alle Restarts benutzen, (fast) so lange

dauern, wie sie wollen, und auch einige reentrant-fähige Unterprogramme des Betriebssystems aufrufen. Die meisten Routinen des Betriebssystems bleiben aber auch weiterhin verboten.

Speziell Ereignisse, die während der Austastlücke bei der Bilddarstellung ausgeführt werden sollen, wird man wohl als asynchrone Events programmieren müssen. Dabei wird man oft sogar nur mit express-asynchroner Behandlung zurechtkommen.

Wird ein normales asynchrones Event aber nicht vom Interrupt-Pfad aus gekickt, sondern durch irgendeine andere Routine, während Interrupts zugelassen sind (via &BCF2 KL EVENT), so wird die zugehörige Behandlungsroutine sofort aufgerufen, ohne daß der Event-Block erst in die Asynchronous Pending Queue eingereiht wird.

Für die praktische Anwendung ist aber die Frage, ob ein Event nun vorher in diese Liste eingetragen wird oder nicht, ziemlich unwichtig, da man davon normalerweise nichts mitbekommt.

Das folgende Beispiel zeigt, wie man einen regelmäßigen Interrupt programmiert, der mit jedem Fast Ticker express-asynchron aufgerufen werden soll:

```
; Einhaengen eines express-asynchronen Events
; mit Near Address auf dem Fast Ticker:
; ----------------------------------------
        ....
        LD   HL,FBLOCK    ; Zeiger auf Fast Ticker Block
        CALL #BCE3        ; KL ADD FAST TICKER aufrufen
        ....
;
; Der Fast Ticker Block:
;
FBLOCK: DEFW #0000        ; Platz fuer Kettungs-Pointer in Fast Ticker
                          ; Chain
        DEFW #0000        ; Platz fuer Kettungspointer in Pending Queue
        DEFB #00          ; Count auf 0 gesetzt
        DEFB %11000001    ; asynchron, express, near address
        DEFW ROUTIN       ; Adresse der Behandlungsroutine
;
ROUTIN: ....              ; Behandlungsroutine
```

Der Fast Ticker Block besteht aus dem Platz für den Kettungs-Pointer in der Fast Ticker Chain und einem Event-Block. Der fängt mit zwei reservierten Bytes für den Kettungs-Pointer in der Asynchronous Pending Queue an. Da es sich aber um ein express-asynchrones Event handelt, bei dem die Routine immer sofort aufgerufen wird, wird dieser Platz nicht beansprucht. Trotzdem muß er bereitgestellt werden.

Der Kernel enthält, zumindest beim CPC 464, hier leider einen Fehler: Das Zählbyte eines express-asynchronen Events wird zwar erhöht, nach Beendigung der Behandlungsroutine aber nicht mehr auf Null zurückgestellt. Dadurch wird beim nächsten Anstoßen dieser Routine nur noch das Zählbyte erhöht (bis maximal 127), die Routine aber nicht mehr ausgeführt! Deshalb muß die Event-Behandlungsroutine ihr Zählbyte immer selbst auf Null zurücksetzen.

Synchrone Events

Nun wäre es aber schade, wenn man all die Unterprogramme der verschiedenen Firmware Packs nicht aufrufen dürfte. Wie soll man z.B. eine mitlaufende Uhr (Textausgabe: Nicht reentrant) oder ähnliches realisieren können?

Aus diesem Grund hat man bei AMSTRAD einen Event-Typ vorgesehen, dessen Ausführung mit dem Hauptprogramm synchronisiert werden kann. Synchrone Events (Bit 7 im Typen-Byte des Event-Blocks = 0) werden vom Kernel mit jedem Kick in die Synchronous Pending Queue eingetragen (außer, wenn sie bereits eingetragen sind) und nicht aufgerufen! In jedem Fall wird das Zählbyte erhöht (aber auch nur bis maximal +127).

Es ist Aufgabe des Hauptprogramms, in regelmäßigen Abständen beim Kernel nachzufragen, ob mittlerweile ein synchrones Event eingetroffen ist und auf seine Abarbeitung wartet. Dazu sollte man die Routine &B921 HI KL POLL SYNCHRONOUS aufrufen, die ganz im RAM liegt und deshalb besonders schnell ist (weil sie keine ROMs ein- und auszublenden hat).

Dieses "Pollen" darf aber nur geschehen, wenn die Event-Routine alle Routinen des Betriebssystems benutzen darf, denn das ist ja der Sinn für den ganzen Umstand. BASIC benutzt für seine Interrupts ausschließlich synchrone Events. Dabei wird immer zwischen zwei BASIC-Statements gepollt und eventuell die Interrupt-Behandlung gestartet. Das BASIC-Programm muß also nicht selbst pollen. Für es erscheinen die Interrupts wieder unvorhersehbar, asynchron.

Das Pollen gestaltet sich leider etwas umständlich. Hier hat man bei AMSTRAD drei Vektoren vorgesehen, die normalerweise immer nur direkt hintereinander aufgerufen werden können. Zumindest ist das der Fall, wenn ein compiliertes oder reines Assemblerprogramm im Speicher abläuft.

Beim BASIC-Interpreter, der einen privaten RETURN-Stapel für das BASIC-Programm unterhält, gestaltet es sich etwas anders, weshalb die Bearbeitung eines synchronen Events im Kernel auf drei verschiedene Vektoren "gestreckt" wurde. Wer aber nicht gerade vorhat, eine eigene Interpreter-Sprache

zu entwickeln, wird mit diesen Problemen kaum konfrontiert werden. Interpreter, die nur einen Stapel kennen (die also die RETURN-Adressen für das Programm auf dem Maschinenstapel ablegen), können aber ebenfalls wie im folgenden Beispiel vorgehen:

```
Pollen und Aufruf einer synchronen Event-Routine in
Maschinencode-Programmen:


; Hauptprogramm:
;        ...
;                        ; nun pollen:
         CALL #B921      ; HI KL POLL SYNCHRONOUS
         CALL C,RUFE     ; CY=1 --> es liegt etwas vor, --> behandeln
         ...
; und weiter im Hauptprogramm
;        ...
;
; Behandlungsroutine fuer ein synchrones Event aufrufen:
;
 RUFE:   CALL #BCFB      ; KL NEXT SYNC  (hole naechstes Event aus der
                         ; Liste)
         RET  NC         ; CY=0 --> es liegt nichts mehr vor, fertig
         PUSH AF         ; alte Prioritaet retten
         PUSH HL         ; Zeiger auf den Event-Block retten
         CALL #BCFE      ; KL DO SYNC (rufe Behandlungsroutine auf)
         POP  HL         ; Zeiger auf Event-Block zurueck
         POP  AF         ; alte Prioritaet zurueck
         CALL #BD01      ; KL DONE SYNC (Zaehlerbyte zurueckstellen &
                         ; Prioritaet restaurieren)
         JR   RUFE       ; Schleife
```

Synchronen Events wird im Typen-Byte des Event-Blocks jeweils eine Priorität zugeordnet: Diese ist in den Bits 1 bis 4 gespeichert und sollte nicht null sein (das ist nämlich die Priorität des Hauptprogramms).

Außerdem sind alle express-synchronen Events dringender als alle normalen. Mit den Vektoren &BD04 KL EVENT DISABLE und &BD07 KL EVENT ENABLE kann die Behandlung von normalen synchronen Events kurzzeitig ausgesetzt werden. In BASIC sind diese beiden Vektoren über die Befehle DI und EI zugänglich. Alle Unterbrechungen in BASIC sind "normal", nur ON BREAK GOSUB ist "express".

Beim Aufruf von &B921 HI KL POLL SYNCHRONOUS und &BCFB KL NEXT SYNC meldet der Kernel nur solche Events weiter, die eine höhere als die gerade aktuelle Priorität haben.

Das heißt: Der Kernel sortiert die gekickten Events nicht nur entsprechend ihrer Priorität in der Synchronous Pending Queue und zieht Ereignisse höherer Priorität vor bereits wartende, unwichtigere Events vor. Auch bei der

Nachfrage, ob in der Synchronous Pending Queue ein Eintrag wartet, werden nur noch Events höherer Priorität weitergemeldet. Dadurch können Interrupt-Routinen ihrerseits ebenfalls die Warteschlange pollen, um sich von später eingetroffenen wichtigeren Ereignissen unterbrechen zu lassen.

Andere Interrupt-Quellen – External Interrupts

Um einen Interrupt für die Sound-Programmierung zu definieren, muß man dem Vektor &BCB0 SOUND ARM EVENT einen Event-Block übergeben.

Der Break-Mechanismus des Key Managers benutzt einen festen Event-Block, den man nicht ändern kann. Um ein Break-Event zu programmieren, muß man dem Vektor &BB45 KM ARM EVENT nur die FAR ADDRESS der gewünschten Behandlungsroutine mitteilen. Der Vektor trägt diese Adresse in den Event-Block ein und aktiviert ihn.

Eigene Interrupt-Quellen lassen sich mit &BCF2 KL EVENT konstruieren. Dieser Vektor kann sowohl aus dem laufenden Programm als auch vom Interrupt-Pfad aus aufgerufen werden. Diese Routine läßt keine Interrupts wieder zu, wenn sie beim Aufruf verboten waren. Da der Vektor zu dieser Routine aber mit einem Restart gebildet ist, der seinerseits Interrupts zulassen würde, muß man die Routinenadresse aus dem Vektor herausholen, wenn man diese Routine bei ausgeschalteten Interrupts aufrufen will.

Ein sinnvolles Einsatzgebiet für KL EVENT ist der External Interrupt. Wenn hier nicht extrem schnelle Antworten benötigt werden (wofür der Schneider-Interrupt-Mechanismus insgesamt nicht geeignet ist), empfiehlt es sich eigentlich immer, den Hardware-Interrupt in ein Event umzuwandeln. Hier ist man dann viel flexibler und kann alle Optionen ausnutzen, die vom Kernel angeboten werden: normal- oder express-asynchrone und auch synchrone Events.

Das folgende Programm ist ein Beispiel, wie man einen Event-Block vom External Interrupt aus kicken kann. Die Routine liegt im RAM. Bei der Initialisierung wird zunächst die Adresse von KL EVENT aus dem Vektor geholt und in den eigenen Aufruf kopiert. Dadurch erfolgt der Aufruf nachher nicht mehr über einen Restart, sondern direkt. Da bei der Bearbeitung eines External Interrupts das untere RAM immer eingeblendet ist, muß man für KL EVENT das untere ROM aber vorher explizit einblenden.

Dann wird der External Interrupt Entry vorschriftsmäßig gepatcht, und mit einem OUT-Befehl werden Interrupts von der eigenen Erweiterung zugelassen.

Die Interrupt-Routine schaltet zuerst das untere ROM ein und testet dann, ob die Unterbrechung auch von der eigenen Hardware kommt, und verzweigt gegebenenfalls zur Kopie des alten External Interrupt-Eintrags. Andernfalls wird der Event-Block gekickt und die Interrupt-Behandlung beendet. Das Event ist synchron. Bei seiner Behandlung werden keine weiteren Interrupts blockiert, und die Behandlungsroutine selbst kann auf fast alle Firmware-Routinen zugreifen:

```
; Kicken eines Events von einem External Interrupt aus:
; -----------------------------------------------------

INIT:   LD   HL,(#BCF2 + 1)    ; Adresse aus Vektor KL EVENT holen
        RES  7,H               ; ROM-Status-Bits fuer Vektor-Restart
                               ; loeschen
        RES  6,H
        LD   (KICK + 1),HL     ; und in den eigenen Aufruf kopieren

        LD   HL,#003B  ; alten Eintrag am External Interrupt Entry retten
        LD   DE,KOPIE
        LD   BC,5
        LDIR

        LD   HL,PATCH  ; Sprung zur eigenen Interrupt-Routine
                       ; installieren
        LD   DE,#003B
        LD   BC,3
        LDIR

        LD   BC,#wxyz  ; der Hardware mitteilen, dass sie jetzt
                       ; Interrupts
        LD   A,#uv     ; erzeugen darf
        OUT  (C),A
        RET

KOPIE:  DEFS 5         ; Platz fuer Kopie des alten External
                       ; Interrupt-Eintrags
PATCH:  JP   XROUT     ; Patch fuer den External Interrupt

;  Behandlungsroutine fuer den External Interrupt: (über # 400!)
; -------------------------------------------------------------
XROUT:  EX   AF, AF'   ; aktueller ROM-Status und
        LD   C, A      ; Bildschirm-Modus ist in A'
        EX   AF, AF'   ; versteckt →nach C laden.
        RES  2, C      ; Bit 2 beeinflußt unteres ROM
        LD   B, #7F    ; Port-Adresse der ULA
        OUT  (C), C    ; unteres ROM einschalten
        LD   BC,#wxyz  ; testen, ob Interrupt von der eigenen Hardware-
        IN   A,(C)     ; Erweiterung stammt und Interrupt gegebenenfalls
        CP   #uv       ; abstellen
        JR   NZ,KOPIE  ; nicht von eigener Hardware? -> Kopie anspringen

        LD   HL,EVBLOK ; sonst Zeiger auf den Event-Block laden und
KICK:   JP   #0000     ; KL EVENT direkt anspringen -> Event-Block kicken
```

```
EVBLOK: DEFW #0000      ; Platz fuer Hangel-Pointer in der Pending Queue
        DEFB 0          ; Count
        DEFB %00001111 ; Typ: synchron, normal, Prioritaet 7, near
                        ; address
        DEFW EVROUT     ; Adresse der Event-Routine
;  Behandlungsroutine fuer den daraus erzeugten Software-Interrupt:
; ---------------------------------------------------------------

EVROUT: ....            ; Event-Behandlungsroutine
        ....
```

Abstellen von Interrupts

Weit schwieriger als die Initialisierung eines (regelmäßigen) Ereignisses gestaltet sich das Abstellen. Hier muß man nämlich mehrere Ebenen berücksichtigen:

1. Die Interrupt-Quelle abstellen

Die Fast Ticker Chain und die Frame Flyback Chain des Kernel ermöglichen es zunächst einmal nur, Event-Blocks einzuhängen, die fortlaufend gekickt werden. In der Ticker Chain kann man durch den Reload-Wert Null erreichen, daß der Event-Block nur einmal angestoßen wird. Trotzdem verbleibt auch hier der Ticker Block in der Liste. Eine Bearbeitung der Ticker Chain führt auch weiterhin zu geringen Zeitverlusten, weil sich der Kernel ständig an diesem Block vorbeihangeln muß.

Über die Vektoren &BCDD KL DEL FRAME FLY, &BCE6 KL DEL FAST TICKER und &BCEC KL DEL TICKER kann man die Blocks wieder vollständig aus den Interrupt-verursachenden Listen aushängen. Das kann auch die Event-Behandlungsroutine selbst machen, wenn es nicht gerade ein express-asynchrones Event ist.

Dadurch werden keine weiteren Kicks mehr erzeugt. Es kann aber sein, daß noch einige Kicks auf ihre Ausführung warten. Die würden dann trotzdem noch ausgeführt.

Bei synchronen Events kann man dabei sowohl vom Hauptprogramm als auch von der Behandlungsroutine aus nie sicher sein, daß in der Synchronous Pending Queue nicht noch ein Event wartet, wenn man die Interrupt-Erzeugung abgestellt hat.

Bei asynchronen Events existiert das Problem nur, wenn die Interrupt-Routine sich selbst aushängen will. Das Hauptprogramm hingegen kann sicher sein, daß, wenn es den Block aus der Liste entfernt hat, auch keine Events mehr

warten. Der Kernel übergibt nach einem (Hardware-) Interrupt die Kontrolle ja erst dann wieder an das Hauptprogramm zurück, wenn alle gekickten, asynchronen Events abgearbeitet sind und die Asynchronous Pending Queue leer ist.

Es kann nur sein, daß die asynchrone Routine so oft gekickt wird und so viel Rechenzeit verbraucht, daß das Hauptprogramm überhaupt nicht mehr zum Zug kommt!

Break- und Sound-Events sind von Natur aus nur "one shots", so daß sich hier ein Abstellen erübrigt. Vielmehr lebt man hier in ständiger Sorge, daß die Event-Blocks auch immer wieder aktiviert werden, wenn sie einmal angestoßen wurden.

2. Behandlung abstellen

Sollen auch noch die ausstehenden Kicks ignoriert werden, muß man die Event-Blocks selbst ruhigstellen.

Bei asynchronen Events tritt dieses Problem auf, wenn das die Interrupt-Routine selbst machen will. Das kann nötig sein, wenn ein "one shot" programmiert werden soll.

Express-asynchrone Events können sich selbst nur ruhigstellen, indem sie das Zähl- und Steuerbyte im Event-Block auf einen negativen Wert (–64) setzen. Normale asynchrone Events können alternativ dazu den Vektor &BD0A KL DISARM EVENT aufrufen, der aber nichts anderes macht. Dadurch können zwar auch weiterhin Kicks eintreffen. Sie führen aber nicht mehr dazu, daß das Event "erfolgreich" angestoßen wird. Die eintreffenden Kicks werden einfach ignoriert.

Bei synchronen Events sieht die Sache etwas anders aus. Hier reicht es nicht, den Event-Block durch Negativ-Setzen des Zählbytes kaltzustellen.

Wenn sich der Event-Block zu diesem Zeitpunkt in der Synchronous Pending Queue befindet, wird er beim nächsten Pollen trotzdem aufgerufen. Der Kernel testet nicht, ob ein Block, der in dieser Warteschlange eingetragen ist, nicht vielleicht abgeschaltet wurde.

Hier muß man den Vektor &BCF8 KL DEL SYNCHRONOUS aufrufen, der den Event-Block ruhigstellt (indem er das Zählbyte auf -64 setzt) und gegebenenfalls auch aus der Synchronous Pending Queue entfernt.

3. Sukzessive Initialisierung

Bevor ein Event-Block erneut initialisiert werden darf, muß er abgeschaltet und aus seiner Pending Queue entfernt sein. Sonst kommt der Kernel mit seinen Listen durcheinander. Bei synchronen Events kann das die Event-Routine selbst machen, wenn sie vorher KL DEL SYNCHRONOUS aufruft.

Bei asynchronen Events geht das jedoch nicht, weil KL DISARM EVENT nur das Zählbyte auf -64 setzt, um den Block kaltzustellen. Der Block kann durchaus noch in der Asynchronous Pending Queue sein. Asynchrone Events können also nur durch das Hauptprogramm erneut initialisiert werden.

Wenn man den Event-Block selbst mit neuen Werten beschickt, muß man sicherstellen, daß er in der Zwischenzeit nicht gekickt werden kann. Am besten hängt man dazu den Event-verursachenden Block aus der entsprechenden Ticker Chain aus. Wenn das nicht geht, oder wenn Ihnen das zu umständlich ist, müssen Sie darauf achten, daß das Zählbyte als letztes auf Null gesetzt oder für diese Zeit der Interrupt verboten wird.

Drucker-Spooler

Als Beispiel für diese doch recht komplexe Materie dient ein per Software realisierter Drucker-Spooler. Dieses Programm hat die Aufgabe, Wartezeiten bei der Textausgabe auf dem Drucker zu überbrücken.

Dazu benutzt der Spooler einen Pufferspeicher, in den das laufende Programm auf der einen Seite hineinschreibt und der auf der anderen Seite via Interrupt ausgelesen und zum Drucker geschickt wird. Dadurch kann das laufende Programm, wenn der Drucker beschäftigt ist, trotzdem weiterschreiben. Die Zeichen werden vom Spooler nur erst einmal in seinem Puffer zwischengespeichert.

Gepatcht wird hier zunächst einmal die Indirection IND MC WAIT PRINTER, die normalerweise den Status der Busy-Leitung vom Drucker testet und ein Zeichen ausdruckt, sobald dieser bereit wird. Spätestens nach ca. 0,4 Sekunden soll sie aber auch unverrichteter Dinge zurückkehren.

Zum Ausdruck werden letztlich die beiden Vektoren MC BUSY PRINTER (Busy-Test) und MC SEND PRINTER (Sende Zeichen ohne Busy-Test) aufgerufen. MC PRINT CHAR, der Vektor, der beide Funktionen ineinander vereint, kann nicht benutzt werden, da dieser nur die gepatchte Indirection selbst wieder aufrufen würde!

Der alte Eintrag in der Indirection kann aber auch nicht mitbenutzt werden, weil der Ausdruck via Interrupt erfolgen soll und hier Wartezeiten von 0,4 Sekunden, wenn der Drucker gerade eine Zeile druckt, das System lahmlegen würden. Und das soll durch den Spooler ja gerade verhindert werden.

Da beim Aufruf einer Event-Routine das untere ROM immer eingeblendet ist, werden die beiden Routinen nicht über ihre Vektoren, sondern über eine Kopie, aus der der Restart entfernt wurde, aufgerufen. Dadurch wird Rechenzeit gespart.

Der Puffer ist, wenn man sich an die Länge im Listing hält, etwas über 1000 Zeichen lang und überschreibt die Initialisierungsroutine. Die wird ja nur einmal aufgerufen und ist danach uninteressant.

Wird nun vom laufenden Programm ein Zeichen zum Drucker geschickt, so führt das zum Aufruf der Routine JOB. Die Ein/Ausgabebedingungen, die von JOB befolgt werden müssen, sind von der Indirection bestimmt:

- DE, HL, IX, IY dürfen nicht verändert werden und
- ein gesetztes CY-Flag zeigt an, daß das Zeichen erfolgreich abgesetzt wurde.

Der Pufferspeicher ist programmtechnisch gesehen natürlich eine Queue, die als Ringspeicher in einem Array für Fixed Length Records realisiert ist. Verwaltet wird er von den beiden Zeigern ANF und ENDE. ANF zeigt dabei immer auf das erste (älteste) Zeichen im Puffer, das als nächstes gedruckt werden muß, und ENDE auf den ersten freien Platz im Puffer, an dem das nächste eintreffende Zeichen gespeichert werden kann. Der zusätzliche Aufwand, einen Zeiger in einem Ringspeicher weiterzustellen, ist in dem Unterprogramm INCRBC zusammengefaßt.

Zwei "besondere" Zustände müssen beim Speichern erkannt werden:

1. Der Puffer ist noch leer. Dann muß der regelmäßige Interrupt gestartet werden, der die Zeichen wieder aus dem Puffer ausliest und zum Drucker schickt.

2. Der Puffer ist voll. Dann muß die Routine warten, bis via Interrupt ein Zeichen weggedruckt und ein Platz frei wird.

Diese beiden Zustände sind gar nicht so leicht auseinanderzuhalten: Ist der Puffer leer, so zeigen ANF und ENDE auf dieselbe Position. Zieht man sie voneinander ab, erhält man Null (auf diese Weise wird das im Programm getestet). Ist der Puffer ganz voll, so wurde ANF einmal durch den gesamten Ringspeicher rundgedreht und steht wieder an seinem alten Platz!

Die einfachste Methode, dieses Problem zu umgehen, ist, den Puffer bereits per Definition als voll zu erklären, wenn de facto noch ein Platz frei ist. So wird das auch in diesem Programm gemacht. Um zu testen, ob der Puffer leer ist, muß man ANF und ENDE vergleichen, bevor man ENDE weiterstellt. Um zu testen, ob der Puffer voll ist, muß man ANF und ENDE vergleichen, nachdem man ENDE erhöht hat.

War der Puffer vorher leer, so wird ein Block auf dem Universal-Ticker eingehängt, der beim nächsten Ticker-Interrupt zum Aufruf von EVROUT führt. Der Event-Block ist normal, asynchron mit near address. Letzteres ist sinnvoll, da EVROUT im RAM liegt und zwei Routinen im unteren ROM aufrufen muß. Die Standardeinstellung beim Aufruf von Event-Routinen (unten ROM, oben unbekannt) ist also genau richtig. Asynchron ist man vom Pollen des BASIC-Interpreters unabhängig, andererseits aber nur normal, weil so EVROUT keine weiteren Hardware-Interrupts blockiert und MC SENC CHAR aufrufen kann. Diese Firmware-Routine läßt nämlich Interrupts wieder zu (auch, wenn sie nicht über Restart aufgerufen wird)!

Ist der Puffer voll, so wird nicht nur gewartet, daß per Interrupt ein Zeichen gedruckt wird, sondern ständig aktiv gekickt, weil, es jetzt eilig ist! EVROUT kann man aber nicht direkt aufrufen, weil dann ein zufälliger Parallelaufruf vom Ticker nicht auszuschließen ist! Wird das Event aber systemkonform über KL EVENT gekickt, so erhöht ein parallel erscheinender Kick vom Ticker nur das Zählbyte im Event-Block. EVROUT wird zwar einmal mehr, aber "nacheinander" aufgerufen!

EVROUT selbst ist leichter verständlich. Zunächst wird getestet, ob der Drukker überhaupt ein Zeichen empfangen kann, und nur dann weiter gemacht. Das Zeichen wird aus dem Puffer geholt und gedruckt. Der Zeiger ANF wird weitergestellt und abgespeichert. Dann wird getestet, ob der Puffer leer ist und in diesem Fall der Event-Block entschärft (Count := –64 = &C0), und der Ticker Block ausgehängt.

Wenn nicht, wird der Kick-Zähler in diesem Fall auf 20 erhöht! Die Eventroutine simuliert für sich selbst 20 weitere angeblich in der Zwischenzeit eingegangene Kicks. Das führt dazu, daß sie sofort nach ihrem Abschluß mit RET vom Kernel erneut 20mal aufgerufen wird. Jetzt wird der Drucker zunächst aber nur busy sein, EVROUT kehrt jedesmal sofort zurück. Mußte der Drukker das Zeichen aber nur intern abspeichern und noch nicht ausdrucken, so wird er wieder empfangsbereit, bevor EVROUT das zwanzigste Mal aufgerufen wird. Dann wird EVROUT erneut ein Zeichen los, und alles beginnt von vorne. Das geht so lange, bis entweder der Puffer leer ist oder der Drucker endlich eine Zeile druckt und mehr als 20mal nacheinander busy bleibt.

Wer einen langsamen Drucker hat, kann den Selbstkick-Wert noch weiter erhöhen. Erkennbar wäre das daran, daß der Drucker zwischen dem Ausdruck zweier ganz normaler Textzeilen eine Pause von einer Sekunde oder länger einlegt.

Durch diesen Trick wird man pro Interrupt nicht nur ein Zeichen, sondern gleich eine ganze Zeile los. "Normale" CPC-Software-Spooler benutzen den Fast Ticker und senden (maximal) pro Sekunde 300 einzelne Zeichen. Speziell bei Grafiken umfaßt eine Zeile aber mehr als 640 einzelne Bytes! Hier würde der Spooler zum Klotz am Bein, weil mit der Methode "Fast Ticker" allein das Übermitteln einer Zeile über zwei Sekunden dauerte!

Semaphoren

Der Drucker-Spooler hat in dieser "Bauart" allerdings ein Problem, das man vielleicht nicht sofort erkennt: Weil sich die Event-Routine selbst abschaltet, muß man aufpassen, daß man sie immer wieder kickt, wenn der Timer "abgelaufen" ist. Andererseits darf man sie aber nicht zuviel kicken, weil die Event-Routine nicht testet, ob der Puffer leer ist.

Das führte dann auch zu dem kurzzeitigen Verbieten des Interrupts beim Label JOB1: Es könnte (rein theoretisch) passieren, daß genau zwischen LD (ENDE),BC und LD HL,(ANF) das Event ausgelöst wird, wenn man den Interrupt nicht verboten hätte. Dann könnte die Interrupt-Routine gerade alle Zeichen bis auf das neue letzte Zeichen ausdrucken, bevor der Drucker wieder zu lange busy bleibt. Im Puffer wäre noch ein Zeichen enthalten, ANF stände einen Platz vor dem neuen ENDE, mithin also genau auf der Position vom alten ENDE!

Der darauf folgende Vergleich ergäbe, daß ANF=ENDEalt ist und damit, daß der Puffer vorher leer gewesen sein muß und daß der Ticker Block eingehängt werden muß. Obwohl ein noch aktiver Ticker Block eingehängt würde, wäre das nicht weiter schlimm (der Kernel erkennt das), wenn nicht in der Zwischenzeit das Event erneut gekickt wird und erfolgreich das (nun tatsächlich) letzte Zeichen absetzen kann. Dann würde der Ticker Block von JOB erneut eingehängt, wenn der Puffer bereits leer ist. Da die Routine EVROUT nicht auf einen leeren Puffer testet, würde ANF den ENDE-Zeiger überholen und der leere Puffer einmal zusätzlich ausgedruckt.

Dieser Fall ist real zwar höchst unwahrscheinlich, gleichwohl "möglich" und muß ausgeschlossen werden. Das wird durch das kurze Verbieten eines Interrupts erreicht. Wird jetzt vor JOB1 das (alte) ENDE erreicht, so hängt sich die Event-Routine aus und muß für das neue letzte Zeichen wieder angestoßen werden.

Wird das alte ENDE erst nach EI erreicht, so geht es in den Test nicht mehr ein. Jetzt kann man sich überlegen, was passiert, wenn zufällig ANF bis ENDEalt gestellt würde: Der Ticker Block wird nicht eingehängt, bleibt aber aktiv (weil das von EVROUT erkannte ENDE jetzt schon ENDEneu ist). Wird ANF aber schon bis ENDEneu weitergestellt, so wird der Ticker Block wieder ausgehängt. Das ist auch o.k., weil der Puffer dann ja auch tatsächlich leer ist.

Ganz allgemein ist die Programmierung von parallel laufenden Prozessen außerordentlich viel schwieriger als ein einziger, linear ablaufender Strang. Dieses Beispiel wird zwar dadurch kompliziert, daß sich die Interrupt-Routine selbst wieder aushängen kann. Aber diese Art von Problemen tritt bei paralleler Verarbeitung immer wieder auf und hat in der Fachliteratur auch einen eigenen Namen: das Semaphoren-Problem.

Semaphoren sind Signale, mit denen beispielsweise ein Programm einem parallel laufenden Prozeß einen Zugriff auf eine bestimmte Abteilung des Betriebssystems, Daten o.ä. signalisieren kann. Dabei ist es ausgesprochen wichtig, daß das Lesen und Neu-Setzen des Signals nicht von einem Parallelprozeß unterbrochen werden kann. Dieser Zugriff muß unteilbar sein. Sonst wäre der Fall denkbar, daß Prozeß 1 auf ein Flag zugreift, sieht, er darf und wird, bevor er das Flag auf "besetzt" stellen kann, von einem weiteren Prozeß unterbrochen, der das Flag ebenfalls testet. Dann erkennen beide, daß sie einen, wie auch immer gearteteten Zugriff machen dürfen. Der Zweck der Semaphore wäre nicht erreicht.

Entsprechend ist es bei JOB1: Der Test, ob der Puffer im Augenblick leer ist, und das endgültige Anfügen des neuen Zeichens müssen "unteilbar" durchgeführt werden.

```
; Interrupt-Programmierung in Maschinensprache:
; ----------------------------------------------
;
; ein Druckerspooler   vs. 3.6.86   (c) G.Woigk
; ------------------   ----------   -----------

        ORG  40000

ADDTIK: EQU  #BCE9          ; KL ADD TICKER
DELTIK: EQU  #BCEC          ; KL DEL TICKER
EVENT:  EQU  #BCF2          ; KL EVENT
BUSY:   EQU  #BD2E          ; MC BUSY PRINTER
SEND:   EQU  #BD31          ; MC SEND PRINTER
IPRINT: EQU  #BDF1          ; IND MC WAIT PRINTER
;
; Initialisierung
; ---------------
;
BANF:   EQU  $              ; Druckerpuffer-Anfang ist HIER!
```

```
INIT:   LD   HL,PATCH        ; Sprung zur eig. Routine JOB in die
        LD   DE,IPRINT       ; Indirection kopieren
        LD   BC,3
        LDIR

        LD   HL,(BUSY+1)     ; MC BUSY PRINTER ohne Restart an
                             ; Aufrufstelle
        RES  7,H             ; kopieren
        RES  6,H
        LD   (EVROUT+1),HL

        LD   HL,(SEND+1)     ; MC SEND PRINTER ohne Restart an
                             ; Aufrufstelle
        RES  7,H             ; kopieren
        RES  6,H
        LD   (PRINT+1),HL
        RET

        DEFS 1000            ; *** Puffer ***

PATCH:  JP   JOB             ; Patch fuer PRINT

BENDE:  EQU  $               ; Druckerpuffer-Ende ist HIER!  (last+1)
ANF:    DEFW BANF            ; Zeiger auf 1. Zeichen (next to print)
ENDE:   DEFW BANF            ; Zeiger hinter letztes Zeichen (next free
                             ; place)

TICKB:  DEFS 6               ; TICKER BLOCK:
EVBLOK: DEFS 2               ; und der darin enthaltene Event-Block:
COUNT:  DEFB 0               ; COUNT: Zaehl- und Steuerbyte
        DEFB %10000001       ; CLASS: asynchron, normal, near address
        DEFW EVROUT          ; Routinenadresse
;
; BC innerhalb des Puffers weiterstellen
; --------------------------------------
;
INCRBC: LD   HL,BENDE
        AND  A
        INC  BC
        SBC  HL,BC
        RET  NZ
        LD   BC,BANF
        RET
;
; Ersatzroutine fuer IND MC WAIT PRINTER: Zeichen im Puffer speichern
; -------------------------------------
;
JOB:    PUSH HL              ; retten wg. I/O-Bedingungen
        LD   BC,(ENDE)
        LD   (BC),A          ; Zeichen im Puffer speichern
        PUSH BC
        CALL INCRBC          ; ENDE-Zeiger weiterstellen

JOB2:   LD   HL,(ANF)        ; Puffer voll?   (ANF = ENDEalt?)
        AND  A
```

```
        SBC  HL,BC
        JR   NZ,JOB1          ; NEIN --> weiter

; Puffer ist voll. Solange direkt kicken, bis wieder Platz ist.
; (evtl. koennte man hier einen Test auf 'time out' einfuegen)

        PUSH BC               ; Event kicken:
        PUSH DE
        LD   HL,COUNT
        LD   (HL),0
        LD   HL,EVBLOK
        CALL EVENT            ; und anstossen (wird sofort aufgerufen)
        POP  DE
        POP  BC
        JR JOB2               ; und testen, ob was gedruckt wurde

JOB1:   DI
        LD   (ENDE),BC        ; ENDE-Zeiger abspeichern: Erst jetzt ist A
                              ; drin!

        LD   HL,(ANF)         ; Test: War Puffer vorher leer? (ANF =
                              ; ENDEalt?)
        EI
        POP  BC               ; alter Wert von ENDE ohne dieses Zeichen
        AND  A
        SBC  HL,BC

        POP  HL               ; HL restaurieren
        SCF                   ; CY := 1 -> o.k. anmerken
        RET  NZ               ; zurueck, wenn Puffer vorher nicht leer war

; Puffer war vorher leer, dann Ticker Block einhaengen:

        PUSH HL               ; sonst: Ticker Block einhaengen:
        PUSH DE
        LD   HL,COUNT
        LD   (HL),0
        LD   HL,TICKB         ; HL -> Ticker Block
        LD   DE,1             ; Count Down
        LD   BC,1             ; Reload Count
        CALL ADDTIK           ; Ticker Block einhaengen
        POP  DE
        POP  HL
        SCF                   ; CY := 0 -> o.k. anmerken
        RET
;
; Event-Routine: Drucke Zeichen aus dem Puffer:
; ---------------------------------------------
;
EVROUT: CALL #00              ; MC BUSY PRINTER: Drucker bereit?
        RET  C                ; nein, dann fertig

        LD   BC,(ANF)
        LD   A,(BC)           ; Zeichen aus Puffer holen
PRINT:  CALL #00              ; MC SEND CHAR: zum Drucker senden
```

```
CALL INCRBC          ; ANF-Zeiger weiterstellen
LD   (ANF),BC        ; und wieder abspeichern

LD   HL,(ENDE)       ; ANF- mit ENDE-Zeiger vergleichen:
AND  A
SBC  HL,BC           ; Z=1 -> Puffer leer?

LD   HL,COUNT        ; vorsorglich 20 weitere Kicks anmerken
LD   (HL),20         ; falls noch was im Puffer ist, wird nach
                     ; dem
RET  NZ              ; Return EVROUT noch 20mal aufgerufen

LD   (HL),#C0        ; Puffer leer -> Event-Block ruhigstellen
LD   HL,TICKB        ; und den Ticker Block aushaengen
JP   DELTIK
```

DIE BASIC-VEKTOREN

Nach den Kernel-Routinen im Main Firmware Jumpblock folgt noch das Machine Pack mit einigen Hardware-nahen Routinen etc. und der Vektor JUMP RESTORE. Darüber sind nur noch Vektoren, die nicht mehr zum Betriebssystem gezählt werden: die BASIC-Vektoren.

Da diese aber so nützliche Dinge wie die Fließkomma-Arithmetik und einen Vektor zum Zeileneditor beinhalten, wird man auf sie wohl kaum verzichten wollen. Auch diese Vektoren werden mit jedem System-RESET und jedem Aufruf von JUMP RESTORE im RAM eingerichtet. Daß sie nicht zum Betriebssystem gehören, hat aber einen ganz entscheidenden Nebeneffekt: Diese Vektoren haben keine feste Lage im RAM.

Arithmetik-Vektoren

Blieb man bis hierhin vor Kompatibilitätsproblemen weitgehend verschont, so ist das jetzt anders. Alle BASIC-Vektoren liegen bei allen drei CPCs an einer anderen Stelle! Das liegt zum einen daran, daß die BASIC-Vektoren direkt an den Main Firmware Jumpblock angefügt wurden und dieser bei allen drei CPCs um einige Vektoren verlängert wurde (hauptsächlich Grafik-Vektoren und beim CPC 6128 der RAM-Select-Vektor).

Zum anderen hat man den Arithmetikteil beim CPC 664/6128 erheblich gestrafft: Einige der Fließkomma-Vektoren fielen heraus, und die Integer-Rechenroutinen, die beim CPC 464 noch im unteren ROM lagen, wurden beim CPC 664 und 6128 ins BASIC-ROM integriert. Diese Vektoren wurden damit auch überflüssig. Eigentlich war das nur zu befürworten, da Rechnungen mit

Integer-Zahlen dadurch noch schneller waren. Nur Assemblerprogrammierer stehen vor dem Problem, daß sie jetzt die Rechenroutinen selbst mit RST 3 aufrufen müssen.

Wer nur für den eigenen Gebrauch programmiert, hat es noch am leichtesten. Er muß sich nur an die im Anhang für seinen Computer angegebenen Adressen halten. Beim CPC 664 und 6128 muß man für Integer-Rechnungen die entsprechenden Routinen im BASIC-ROM mit einem Restart direkt aufrufen.

Werden umfangreichere Berechnungen mit Integer-Zahlen durchgeführt, kann man auch mit den Vektoren HI KL U ROM ENABLE und HI KL U ROM DISABLE für die Dauer der Integer-Operationen das BASIC-ROM ständig einblenden und dann die entsprechenden Routinen mit einfachen CALL-Befehlen aufrufen:

```
; Integer-Berechnungen beim CPC 664/6128
;
        ...
        CALL #B900    ; HI KL U ROM ENABLE
        ...
        ...           ; hier Berechnungen
        ...
        CALL #B903    ; HI KL U ROM DISABLE
        ...
```

Bei dieser Methode muß nur immer sicher sein, daß im obersten Adreßviertel auch wirklich das BASIC-ROM selektiert ist. Normalerweise (bei der Programmierung von Utilities für BASIC o.ä.) wird das aber immer der Fall sein.

Entsprechend können übrigens auch CPC-464-Routinen mit dem unteren ROM verfahren. Mit den Vektoren HI KL L ROM ENABLE und HI KL L ROM DISABLE (&BC06 und &BC09) kann man das untere ROM einblenden. Wenn man sich dann die Adressen der Routinen einmal aus den Vektoren herausholt, kann man sie immer direkt aufrufen und auch hier die Bankschalterei umgehen.

Wer aber Programme schreibt, die nachher auch auf den "kompatiblen" Geschwistern laufen sollen, der kommt nicht umhin, entweder drei verschiedene Versionen zu schreiben oder zuallererst zu testen, welche Version er jeweils vorliegen hat. Dazu kann man den Vektor &B915 HI KL PROBE ROM benutzen und dann vielleicht eine eigene Sprungleiste entsprechend initialisieren.

Der Vorteil dieser BASIC-Vektoren ist, daß man beim Rechnen mit Fließkommazahlen in Assembler die einzelnen Vektoren wieder als Module auffassen kann, von denen man nur die Schnittstellenbeschreibung zu kennen braucht: Eingabe – Funktion – Ausgabe.

Das folgende Beispiel demonstriert eine Fließkomma-Multiplikation in Assembler:

```
; FLO-Multiplikation
; ------------------
;
       ORG  40000     ; Parameter von einem BASIC-Aufruf uebernehmen
       LD   L,(IX+0)  ; HL = zweiter Parameter
       LD   H,(IX+1)
       LD   E,(IX+2)  ; DE = erster Parameter
       LD   D,(IX+3)
       CALL #BD61     ; CPC 464: #BD61 / 664: #BD82 / 6128: #BD85
       RET            ; FLO(HL) := FLO(HL) * FLO(DE)
```

```
100 ' BASIC-Aufruf dazu:
110 '
120 DIM dummy%(8000),a!(2)
120 a!(1)=12345.6
130 a!(2)=98765.4
140 '
150 CALL 40000,@a!(1),@a!(2)
160 '
170 PRINT a!(2)
```

Mit dummy%(8000) wird ein Array definiert, das etwas über 16000 Bytes lang ist. Das nachfolgend definierte Feld a!(2) wird über dummy%() angelegt und liegt deshalb im zentralen RAM.

Da die Fließkomma-Routinen im unteren ROM untergebracht sind, können sie nicht auf Variablen im untersten RAM-Viertel zugreifen. Übergibt man eine Adresse unterhalb von &4000, so wird der entsprechende Wert aus dem ROM gelesen, hinterher das Ergebnis aber an dieser Stelle ins RAM geschrieben (Schreibzugriffe der CPU gehen ja immer an das RAM!). So würde zwar die Zielvariable verändert, ein sinnvolles Ergebnis käme aber nicht heraus.

Mit dem Aufruf in Zeile 150 werden dem Assemblerprogramm die Adressen der beiden Feldelemente a!(1) und a!(2) übergeben.

Diese werden im Assemblerprogramm zunächst nach DE und HL geladen, und dann wird die Multiplikationsroutine aufgerufen. Diese legt das Ergebnis zum Schluß wieder in der durch HL adressierten Variablen ab, die in diesem Beispiel a!(2) ist.

Allgemein wird aber ein Maschinencode-Programm, das Fließkommazahlen übernimmt, diese zuerst einmal in reservierte Bereiche im zentralen RAM kopieren, damit die Rechenroutinen auch ganz sicher darauf zugreifen können.

Das folgende Beispiel greift den Kreis-Algorithmus aus dem Kapitel zur Grafik-VDU auf:

```
200 n%=1+PI*SQR(r)
210 ORIGIN xm,ym
220 dx!=0:dy!=r:MOVE dx!,dy!
230 sinus!=SIN(2*PI/n%)
240 cosin!=COS(2*PI/n%)
250 '
260 FOR n%=1 TO n%
270    z!=dx!*cosin!-dy!*sinus!
280   dy!=dy!*cosin!+dx!*sinus!
290   dx!=z!
300   DRAW dx!,dy!
310 NEXT:RETURN
```

Das entsprechende Assemblerprogramm ist für den CPC 464 geschrieben. Benutzer eines CPC 664 oder 6128 müssen die entsprechenden Adressen den Tabellen im Anhang entnehmen.

An diesem Beispiel sieht man aber auch, zu welchen Längen in Assembler geschriebene Rechenroutinen neigen. Anhand der Kommentare sollte es aber leicht zu verfolgen sein, welcher Wert sich wann in welchem Speicher befindet.

```
;   |CIRCLE,xm,ym,r[,f]          vs. 4.6.86    (c) G.Woigk
;   -------------------          ----------    -----------
;
        ORG  #6000
        JP   KREIS
;
; HIER: Einbindung als RSX & Relocalisation
;
DEGRAD: EQU  #BD73     ; A=0     => RADIANT              Angaben fuer CPC 464!
FLOMOV: EQU  #BD3D     ; FLO(HL) := FLO(DE)              ----------------------
FLOPI:  EQU  #BD76     ; FLO(HL) := PI
FLOSIN: EQU  #BD88     ; FLO(HL) := SIN(FLO)
FLOCOS: EQU  #BD8B     ; FLO(HL) := COS(FLO)
FLOSQR: EQU  #BD79     ; FLO(HL) := SQR(FLO)
FLOADD: EQU  #BD58     ; FLO(HL) := FLO(HL) + FLO(DE)
FLOMUL: EQU  #BD61     ; FLO(HL) := FLO(HL) * FLO(DE)
FLODIV: EQU  #BD64     ; FLO(HL) := FLO(HL) / FLO(DE)
FLOSUB: EQU  #BD5B     ; FLO(HL) := FLO(HL) - FLO(DE)
INTFLO: EQU  #BD40     ; FLO(DE) := HL,A=VZ
FLOINT: EQU  #BD46     ; HL,A=VZ := FLO(HL)
INTVZW: EQU  #BDC7     ; HL  := -1 * HL
;
SETPEN: EQU  #BBDE     ; setze Grafik-Farbstift
SETORG: EQU  #BBC9     ; setze Grafik-Origin
GETORG: EQU  #BBCC     ; erfrage Grafik-Origin
PLOTR:  EQU  #BBED     ; plotte einen Punkt relativ
DRAW:   EQU  #BBF6     ; ziehe Linie absolut
MOVE:   EQU  #BBC0     ; setze Grafik-Cursor absolut
;
SINUS:  DEFS 5         ; Speicher fuer SIN(PI*2/n)
```

```
COSIN:  DEFS 5          ; Speicher fuer COS(PI*2/n)
XPOS:   DEFS 5          ; Speicher fuer X-Koord. des Kreisbogens
YPOS:   DEFS 5          ; Speicher fuer Y-Koord. des Kreisbogens
ZW1:    DEFS 5          ; Zwischenspeicher Nr. 1
ZW2:    DEFS 5          ; Zwischenspeicher Nr. 2
NULL:   DEFB 0,0,0,0,0  ; Konstante: 0
;
; ------------------------------
;
KREIS:  CP   5
        RET  NC         ; Zu viele Argumente
        CP   3
        RET  C
        JR   Z,K1
;
        LD   A,(IX)     ; 4. Parameter: Farbangabe
        INC  IX
        INC  IX
        CALL SETPEN     ; Grafikfarbe setzen
;
K1:     CALL GETORG     ; Rette ORIGIN
        PUSH HL
        PUSH DE
        CALL CIRCLE     ; DO THE JOB
        POP  DE
        POP  HL
        JP   SETORG     ; Restaurieren & RET
;
; ------------------------------
;
CIRCLE: XOR  A
        CALL DEGRAD     ; Winkelmass auf radiant einstellen
;
        LD   D,(IX+5)
        LD   E,(IX+4)   ; 1. Parameter: XM
        LD   H,(IX+3)
        LD   L,(IX+2)   ; 2. Parameter: YM
        CALL SETORG     ; Kreismittelpunkt setzen
;
        LD   H,(IX+1)
        LD   L,(IX+0)   ; 3. Parameter: Radius
        BIT  7,H
        RET  NZ         ; Fehler: Neg. Radius
        PUSH HL
        LD   DE,0
        CALL MOVE       ; Startpunkt der Kreislinie: (0,R)
;
        LD   HL,XPOS    ; (HL)  = XPOS
        LD   DE,NULL    ; (DE)  = NULL
        CALL FLOMOV     ; (HL)  = XPOS := NULL
        POP  HL         ;  HL  = Radius = R
        XOR  A          ; Vorzeichen:= Positiv
        LD   DE,YPOS    ; (DE) = YPOS
        CALL INTFLO     ; (HL) = YPOS := R
;
        EX   DE,HL      ; (DE) = YPOS  = R
        LD   HL,ZW1     ; (HL) = ZW1
        CALL FLOMOV     ; (HL) = ZW1 := R
        CALL FLOSQR     ; (HL) = ZW1 := SQR(R)
```

```
        LD   HL,ZW2     ; (HL) = ZW2
        CALL FLOPI      ; (HL) = ZW2 := PI
        EX   DE,HL      ; (DE) = ZW2  = PI
        LD   HL,ZW1     ; (HL) = ZW1  = SQR(R)
        CALL FLOMUL     ; (HL) = ZW1 := PI*SQR(R)
        CALL FLOINT     ;  HL := ROUND(PI*SQR(R))
        INC  HL         ;  HL := 1+ROUND(PI*SQR(R))

;       ***** HL = L = ECKENZAHL N *****

        PUSH HL         ;  N = Schleifenvariable fuer spaeter
;
        LD   DE,ZW1     ; (DE)  = ZW1
        CALL INTFLO     ; (HL)  = ZW1 := N
        LD   HL,ZW2     ; (HL)  = ZW2  = PI
        LD   DE,ZW2     ; (DE)  = ZW2  = PI
        CALL FLOADD     ; (HL)  = ZW2 := 2*PI
        LD   DE,ZW1     ; (DE)  = ZW1  = N
        CALL FLODIV     ; (HL)  = ZW2 := 2*PI/N
        EX   DE,HL      ; (DE)  = ZW2  = 2*PI/N
        LD   HL,SINUS   ; (HL)  = SINUS
        CALL FLOMOV     ; (HL)  = SINUS := 2*PI/N
        LD   HL,COSIN   ; (HL)  = COSIN
        CALL FLOMOV     ; (HL)  = COSIN := 2*PI/N
        CALL FLOCOS     ; (HL)  = COSIN := COS(2*PI/N)
        LD   HL,SINUS   ; (HL)  = SINUS  = 2*PI/N
        CALL FLOSIN     ; (HL)  = SINUS := SIN(2*PI/N)
;
; SCHLEIFE UEBER N STRECKENZUEGE
;
LOOP:   LD   HL,0
        LD   DE,0
        CALL PLOTR      ; Ersten Punkt der Linie doppelt setzen, falls XOR-Modus
;
        LD   DE,XPOS
        LD   HL,ZW1
        CALL FLOMOV     ; (HL)  = ZW1 := XA ( = ALTES X )
        LD   DE,COSIN   ; (DE)  = COS
        CALL FLOMUL     ; (HL)  = ZW1 := XA*COS
        LD   HL,ZW2
        LD   DE,YPOS
        CALL FLOMOV     ; (HL)  = ZW2 := YA ( = ALTES Y )
        LD   DE,SINUS   ; (DE)  = SIN
        CALL FLOMUL     ; (HL)  = ZW2 := YA*SIN
        EX   DE,HL      ; (DE)  = ZW2 := YA*SIN
        LD   HL,ZW1     ; (HL) = ZW1 = XA*COS
        CALL FLOSUB     ; (HL) = ZW1 := XA*COS-YA*SIN = XN
;
        LD   DE,YPOS
        LD   HL,ZW2
        CALL FLOMOV     ; (HL) = ZW2 := YA
        LD   DE,COSIN   ; (DE) = COS
        CALL FLOMUL     ; (HL) = ZW2 := YA*COS
        EX   DE,HL      ; (DE) = ZW2 = YA*COS
        LD   HL,YPOS    ; (HL) = YPOS
        CALL FLOMOV     ; (HL) = YPOS :=YA*COS
        LD   HL,ZW2
        LD   DE,XPOS
        CALL FLOMOV     ; (HL) = ZW2 := XA
        LD   DE,SINUS   ; (DE) = SIN
```

```
        CALL FLOMUL     ; (HL) = ZW2 := XA*SIN
        EX   DE,HL      ; (DE) = ZW2 := XA*SIN
        LD   HL,YPOS    ; (HL) = YPOS
        CALL FLOADD     ; (HL) = YPOS := YA*COS+XA*SIN = YN
;
        CALL FLOINT     ; HL := YN & A:=VZ
        JR   NC,OVFLOW  ; YN zu gross?
        BIT  7,H
        JR   NZ,OVFLOW
        RLA
        CALL C,INTVZW   ; NEG. => HL := -HL
        PUSH HL         ; YN retten
        LD   DE,ZW1     ; (DE) = ZW1 = XN
        LD   HL,XPOS    ; (HL) = XPOS
        CALL FLOMOV     ; (HL) = XPOS := XN
;
        CALL FLOINT     ; HL := XN
        POP  DE         ; DE := YN
        JR   NC,OVFLOW  ; XN zu gross?
        BIT  7,H
        JR   NZ,OVFLOW
        RLA
        CALL C,INTVZW   ; Negativ => HL := -HL
        EX   DE,HL
        CALL DRAW       ; Zeichne Linie
;
        POP  BC         ; C=Schleifenvariable N
        DEC  C
        PUSH BC
        JP   NZ,LOOP    ; Noch eine Linie
;
OVFLOW: POP  BC         ; Abbruch bei Ueberschreitung des Integer-Formats
        RET
```

Der Editor

Die letzte sehr nützliche Abteilung im Betriebssystem, die auch einen Vektor im BASIC-Jumpblock zugestanden bekam, ist der Zeileneditor.

Hiermit können Texteingaben mit einer Höchstlänge von 255 Zeichen neu eingegeben oder auch verändert, editiert, werden. Jeder, der Eingaben in Assemblerprogrammen entgegennehmen will, braucht sich die Routinen hierfür nicht selbst zu programmieren, sondern kann den Zeileneditor aufrufen. Dabei muß er im HL-Register einen Zeiger auf den Textpuffer übergeben, der 256 Bytes lang sein muß. Der zu editierende Text muß dabei immer mit einem Null-Byte abgeschloßen sein.

Im Anhang ist der Editor genau beschrieben, deshalb folgt an dieser Stelle nur noch ein Anwendungsbeispiel: Eine Funktion, die man in BASIC vermißt, ist ein Befehl, mit dem man Strings nicht nur komplett neu eingeben, sondern auch bestehende Strings editieren kann. Dafür ist dieses Beispiel gedacht.

Der dem Aufruf als Parameter übergebene String wird zunächst in den 256-Byte-Puffer kopiert und dann der Zeileneditor aufgerufen. Dieser funktioniert jetzt ganz normal wie auch bei LINE INPUT. Je nachdem, ob der Anwender die Eingabe mit ENTER oder ESC abschließt, ist beim Rücksprung das CY-Flag gesetzt (o.k.) oder auch nicht.

Wenn man die Eingabe mit ESC abbricht, wird der String-Descriptor nicht auf den neuen String eingestellt, der String bleibt also unverändert. Wird die Eingabe aber mit ENTER abgeschlossen, so wird der Descriptor auf den String im Puffer eingestellt. Dieser muß danach noch in den Memory Pool transferiert werden. Das folgende Beispiel zeigt, wie ein Aufruf in BASIC aussehen müßte:

```
 100 KEY DEF 66,0,140                          ' Break-Taste so umdefinieren, dass
 110  KEY 140,CHR$(252)+CHR$(&EF)+CHR$(27)  ' ein Break erkennbar wird
 120 CALL &BB48                                ' ON BREAK CONT
 ... ...
 ... ...
1000 a$="default.dat"                          ' Vorgabewert in a$
1010 PRINT "Welche Datei laden? >>> ";         ' Dialogtext
1020 |EDIT,@a$                                 ' a$ editieren
1025 IF INKEY$=CHR$(27) THEN 1010              ' Breaks abfangen
1030 a$=a$+""                                   ' a$ in den Memory Pool bringen
.... ...
```

```
; String-Editor    vs. 10.6.86    (c) by G.Woigk
; -------------    -----------    --------------
;
; AUFRUF mit:   |EDIT,@A$
; -----------   ---------
;
        ORG  40000
;
EDITOR: EQU  #BD3A          ; CPC 464: &BD3A / 664: &BD5B / 6128: &BD5E
;                            ; BASIC-Vektor zum Zeileneditor
;
;  *********************************
;        Hier Verschiebung und
;    Einbindung als RSX einfuegen.
;  *********************************
;
EDIT:   DEC  A              ; Teste auf einen Parameter
        RET  NZ             ; Parameter-Error
;
; STRING-LAENGE UND -ADRESSE BESTIMMEN
;
         LD   L,(IX+0)      ; Adresse des String-Descriptors
        LD   H,(IX+1)       ; nach HL holen
        PUSH HL             ; und auf den Stack retten
;
        LD   C,(HL)
        LD   B,0            ; BC := Laenge des Strings
        INC  HL
        LD   E,(HL)
        INC  HL
         LD   D,(HL)        ; DE := Adresse des Strings
;
; STRING IN DEN PUFFER KOPIEREN
```

```
        INC  HL
        LD   D,(HL)       ; DE := Adresse des Strings
;
; STRING IN DEN PUFFER KOPIEREN
;
        EX   DE,HL        ; HL = LDIR-Quelle := Adresse des Strings
        LD   DE,BUFFER    ; DE = LDIR-Ziel := Adresse des 256-Byte-Buffer
        XOR  A
        CP   C            ; Laenge = 0?   Dann kein LDIR!
        JR   Z,P2
        LDIR              ; String in den Puffer kopieren
;
; EDITIEREN DES STRINGS
;
P2:     LD   (DE),A       ; String-Abschlussmarke setzen: Byte 0
        LD   HL,BUFFER    ; Zeiger auf den Puffer nach HL laden
        CALL EDITOR       ; und den Zeileneditor aufrufen
        POP  HL
        RET  NC           ; Falls BREAK: String nicht aendern
        PUSH HL
;
; NEUE STRING-LAENGE BERECHNEN
;
        LD   HL,BUFFER    ; HL zeigt auf den neuen $
        XOR  A
        LD   BC,101H
        CPIR              ; Suche die String-Endmarke
        SUB  C            ; A := String-Laenge
;
; STRING-DESCRIPTOR AUF DEN NEUEN STAND BRINGEN
;
P3:     POP  HL           ; HL := Adresse des String-Descriptors
        LD   DE,BUFFER
        LD   (HL),A       ; String-Laenge in den Descriptor eintragen
        INC  HL
        LD   (HL),E
        INC  HL
        LD   (HL),D       ; und die String-Adresse (Pufferadresse).
        RET
;
BUFFER: DEFS 256
```

Kapitel 4

Die Firmware des Schneider CPC

Ganze Bereiche im RAM dienen als Sprungbrett zu den diversen Routinen im unteren Betriebssystem-ROM.

Dieser Aufwand hat zwei Gründe: Zum einen werden diese Vektoren mit Restarts gebildet, die für die Routine die benötigte ROM- und RAM-Konfiguration einschalten, zum anderen kann nur so eine annähernde Kompatibilität zwischen verschiedenen ROM-Versionen gewahrt bleiben. Anwenderprogramme, die auf die Firmware nur über diese Vektoren zugreifen, laufen auch dann noch, wenn im ROM Änderungen vorgenommen werden.

Die Bezeichnungen der Vektoren sind dem "FIRMWARE MANUAL" entnommen.

THE MAIN JUMPBLOCK
Der zentrale Sprungbereich

THE KEY MANAGER (KM) – DIE TASTATUR-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
BB00	KM INITIALIZE	Initialisiere die Tastatur-Routinen.
BB03	KM RESET	Setze die Tastatur-Routinen zurück.
BB06	KM WAIT CHAR	Warte auf ein (expandiertes) Zeichen von der Tastatur.
BB09	KM READ CHAR	Hole ein (expandiertes) Zeichen von der Tastatur (falls eins vorhanden ist).
BB0C	KM CHAR RETURN	Gib ein Zeichen an den KM zurück.

Adresse	Bezeichnung	kurze Beschreibung
BB0F	KM SET EXPAND	Ordne einem Erweiterungszeichen einen String zu.
BB12	KM GET EXPAND	Hole ein Zeichen aus einem Erweiterungs-String.
BB15	KM EXP BUFFER	Lege den Speicherbereich fest, in dem der KM die Erweiterungs-Strings speichern kann.
BB18	KM WAIT KEY	Warte auf ein (nicht expandiertes) Zeichen von der Tastatur.
BB1B	KM READ KEY	Hole ein (nicht expandiertes) Zeichen von der Tastatur.
BB1E	KM TEST KEY	Teste, ob eine bestimmte Taste gedrückt ist.
BB21	KM GET STATE	Erfrage den Status der SHIFT- und CAPS-LOCKS-Taste.
BB24	KM GET JOYSTICK	Teste den Schaltzustand der Joysticks.
BB27	KM SET TRANSLATE	Lege das Zeichen fest, das eine Taste erzeugen soll, wenn sie ohne CTRL oder SHIFT gedrückt wird.
BB2A	KM GET TRANSLATE	Erfrage die Belegung einer Taste ohne CTRL und SHIFT.
BB2D	KM SET SHIFT	Bestimme das Zeichen einer Taste für SHIFT.
BB30	KM GET SHIFT	Erfrage die Tastenübersetzung mit SHIFT.
BB33	KM SET CONTROL	Bestimme das Zeichen einer Taste für CTRL.
BB36	KM GET CONTROL	Erfrage die Tasten-Übersetzung mit CTRL.

Adresse	Bezeichnung	kurze Beschreibung
BB39	KM SET REPEAT	Lege fest, ob eine Taste "Repeaten", sich also automatisch wiederholen, darf, wenn sie lange genug gedrückt wird.
BB3C	KM GET REPEAT	Erfrage, ob eine Taste "Repeaten" darf.
BB3F	KM SET DELAY	Lege die Verzögerungszeiten beim "Repeaten" fest.
BB42	KM GET DELEAY	Erfrage die Verzögerungszeiten beim "Repeaten".
BB45	KM ARM BREAK	Aktiviere den Break-Mechanismus.
BB48	KM DISARM BREAK	Schalte den Break-Mechanismus aus.
BB4B	KM BREAK EVENT	Löse einen Break-Event aus, wenn der Mechanismus aktiv ist.
BD3A	KM SET LOCKS	Lege den SHIFT- und den CAPS/-LOCK-Status neu fest (nur CPC 664 und 6128).
BD3D	KM FLUSH	Lösche alle bisher aufgelaufenen Zeichen im Tastaturpuffer (nur CPC 664 und 6128!).

THE TEXT VDU (TXT) – DIE TEXTAUSGABE-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
BB4E	TXT INITIALISE	Initialisiere die Text-VDU.
BB51	TXT RESET	Setze die Text-VDU zurück.
BB54	TXT VDU ENABLE	Erlaube, daß Zeichen im aktiven Fenster ausgegeben werden.
BB57	TXT VDU DISABLE	Verbiete die Ausgabe von Zeichen im aktiven Fenster.

Adresse	Bezeichnung	kurze Beschreibung
BB5A	TXT OUTPUT	Drucke ein Zeichen, oder befolge den Controlcode.
BB5D	TXT WR CHAR	Drucke ein Zeichen. Controlcodes erzeugen Sonderzeichen.
BB60	TXT RD CHAR	Versuche, ein Zeichen vom Bildschirm zu lesen.
BB63	TXT SET GRAPHIC	Bestimme, ob der Text im laufenden Fenster auf der Cursor- oder Grafikposition ausgegeben werden soll.
BB66	TXT WIN ENABLE	Lege die Grenzen des aktuellen Textfensters fest.
BB69	TXT GET WINDOW	Erfrage die Grenzen des aktuellen Textfensters.
BB6C	TXT CLEAR WINDOW	Lösche die Anzeige im aktuellen Textfenster.
BB6F	TXT SET COLUMN	Bewege den Cursor des aktuellen Textfensters in die angegebene Spalte.
BB72	TXT SET ROW	Bewege den Cursor des aktuellen Textfensters in die angegebene Zeile.
BB75	TXT SET CURSOR	Lege Zeile und Spalte des Cursors neu fest.
BB78	TXT GET CURSOR	Erfrage Zeile und Spalte des Cursors.
BB7B	TXT CUR ENABLE	Schalte Cursor auf der Benutzerebene ein.
BB7E	TXT CUR DISABLE	Schalte den Cursor auf der Benutzerebene aus.
BB81	TXT CUR ON	Schalte den Cursor auf der Systemebene ein.

Adresse	Bezeichnung	kurze Beschreibung
BB84	TXT CUR OFF	Schalte den Cursor auf der Systemebene aus.
BB87	TXT VALIDATE	Überprüfe, ob sich eine Position innerhalb des Textfensters befindet.
BB8A	TXT PLACE CURSOR	Male einen "Cursor-Fleck" auf die Cursorposition.
BB8D	TXT REMOVE CURSOR	Entferne einen "Cursor-Fleck" von der Cursorposition.
BB90	TXT SET PEN	Lege die Stifttinte für die Buchstaben fest.
BB93	TXT GET PEN	Erfrage die momentane Stifttinte.
BB96	TXT SET PAPER	Lege die Hintergrund-Tinte für die Buchstaben fest.
BB99	TXT GET PAPER	Erfrage die momentane Hintergrund-Tinte.
BB9C	TXT INVERSE	Tausche die Papier- und Stifttinten für das momentan aktuelle Textfenster aus.
BB9F	TXT SET BACK	Lege fest, ob die Buchstaben im Transparentmodus geschrieben werden sollen.
BBA2	TXT GET BACK	Erfrage, ob im aktuellen Textfenster der Transparentmodus eingeschaltet ist.
BBA5	TXT GET MATRIX	Erfrage, an welcher Stelle die Grafikinformation über das Aussehen des angegebenen Buchstaben abgelegt ist.
BBA8	TXT SET MATRIX	Bestimme das Aussehen eines Buchstabens neu.
BBAB	TXT SET M TABLE	Teile der Text-VDU mit, in welchem Speicherbereich die selbstdefinierbaren Zeichenmatrizen abgelegt werden können.

Adresse	Bezeichnung	kurze Beschreibung
BBAE	TXT GET TABLE	Erfrage, wo momentan der Bereich für die selbstdefinierbaren Zeichenmatrizen liegt.
BBB1	TXT GET CONTROLS	Erfrage die Lage der Controlcode-Tabelle.
BBB4	TXT STREAM SELECT	Wähle ein anderes Textfenster an.
BBB7	TXT SWAP STREAMS	Vertausche die Einstellungen und Parameter zweier Textfenster.
BD40	TXT ASK STATE	Erfrage Cursor- und VDU-Status des aktuellen Textfensters nur (CPC 664 und 6128).

THE GRAPHICS VDU (GRA) – DIE GRAFIK-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
BBBA	GRA INITIALISE	Initialisiere die Grafik-VDU.
BBBD	GRA RESET	Setze die Grafik-VDU zurück.
BBC0	GRA MOVE ABSOLUTE	Bewege den Grafik-Cursor zur angegebenen Position.
BBC3	GRA MOVE RELATIVE	Bewege den Grafik-Cursor relativ zur momentanen Position.
BBC6	GRA ASK CURSOR	Erfrage die momentane X- und Y-Koordinate des Grafik-Cursors.
BBC9	GRA SET ORIGIN	Lege den Bezugspunkt für die absolute Cursorpositionierung neu fest.
BBCC	GRA GET ORIGIN	Erfrage den momentanen Grafik-Nullpunkt.
BBCF	GRA WIN WIDTH	Lege den linken und rechten Rand des Grafikfensters fest.

Adresse	Bezeichnung	kurze Beschreibung
BBD2	GRA WIN HEIGHT	Lege den oberen und unteren Rand des Grafikfensters fest.
BBD5	GRA GET W WIDTH	Erfrage den linken und rechten Rand des Grafikfensters.
BBD8	GRA GET W HEIGHT	Erfrage den oberen und unteren Rand des Grafikfensters.
BBDB	GRA CLEAR WINDOW	Lösche die Anzeige im Grafikfenster.
BBDE	GRA SET PEN	Lege die Tinte des Zeichenstiftes neu fest.
BBE1	GRA GET PEN	Erfrage die momentane Tinte des Zeichenstiftes.
BBE4	GRA SET PAPER	Lege die Hintergrund-Tinte für die Grafik-VDU neu fest.
BBE7	GRA GET PAPER	Erfrage die momentane Hintergrund-Tinte.
BBEA	GRA PLOT ABSOLUTE	Setze einen Punkt auf die angegebene Position.
BBED	GRA PLOT RELATIVE	Setze einen Punkt relativ zur momentanen Position des Grafik-Cursors.
BBF0	GRA TEST ABSOLUTE	Teste, welche Tintennummer der Punkt auf der angegebenen Position hat.
BBF3	GRA TEST RELATIVE	Teste, welche Tintennummer ein Punkt relativ zur aktuellen Lage des Grafik-Cursors hat.
BBF6	GRA LINE ABSOLUTE	Ziehe eine Linie von der momentanen Position des Grafik-Cursors zur angegebenen absoluten Position.

Adresse	Bezeichnung	kurze Beschreibung
BBF9	GRA LINE RELATIVE	Ziehe eine Linie zu einer Position relativ zur aktuellen Lage des Grafik-Cursors.
BBFC	GRA WR CHAR	Zeichne einen Buchstaben an der Position des Grafik-Cursors.
BD43	GRA DEFAULT	Stelle die Standardwerte für die verschiedenen Optionen der Grafik-VDU ein (nur CPC 664 und 6128).
BD46	GRA SET BACK	Setze den Hintergrund-Modus der Grafik-VDU (nur CPC 664 und 6128).
BD49	GRA SET FIRST	Setze die Erster-Punkt-Option für die zu zeichnenden Linien (nur CPC 664 und 6128).
BD4C	GRA SET LINE MASK	Lege die Punktmaske für Linien neu fest (nur CPC 664 und 6128).
BD4F	GRA FROM USER	Konvertiere die User-Koordinaten (relativ zum Origin) in Basiskoordinaten (relativ zur linken unteren Ecke) (nur CPC 664 und 6128).
BD52	GRA FILL	Male eine beliebige Fläche aus (nur CPC 664 und 6128).

THE SCREEN PACK (SCR) – DIE BILDSCHIRM-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
BBFF	SCR INITIALISE	Initialisiere das Screen Pack.
BC02	SCR RESET	Setze das Screen Pack zurück.
BC05	SCR SET OFFSET	Verändere den Hardware-Scroll-Offset des Bildschirms.

Adresse	Bezeichnung	kurze Beschreibung
BC08	SCR SET BASE	Verlege den Bildschirm in ein anderes RAM-Viertel.
BC0B	SCR GET LOCATION	Erfrage Scroll-Offset und Speicherviertel des Bildschirms.
BC0E	SCR SET MODE	Lösche den Bildschirm, setze den Scroll-Offset auf Null, und lege den Bildschirmmodus neu fest.
BC11	SCR GET MODE	Erfrage den momentan eingestellten Bildschirmmodus.
BC14	SCR CLEAR	Lösche den ganzen Bildschirm mit der Tinte 0.
BC17	SCR CHAR LIMITS	Erfrage, wie viele Buchstaben, abhängig vom aktuellen Bildschirmmodus, in eine Zeile passen.
BC1A	SCR CHAR POSITION	Konvertiere eine "physikalische" Buchstabenposition in die zugehörige Bildspeicheradresse.
BC1D	SCR DOT POSITION	Konvertiere eine Grafik-Position in "Basis"-Koordinaten in die zugehörige Bildspeicheradresse.
BC20	SCR NEXT BYTE	Bestimme die Adresse des Bytes im Bildschirmspeicher rechts von der angegebenen Adresse.
BC23	SCR PREV BYTE	Bestimme die Adresse des Bytes im Bildschirmspeicher links von der angegebenen Adresse.
BC26	SCR NEXT LINE	Bestimme die Adresse des Bytes im Bildschirmspeicher unter der angegebenen Adresse.
BC29	SCR PREV LINE	Bestimme die Adresse des Bytes im Bildschirmspeicher über der angegebenen Adresse.

Adresse	Bezeichnung	kurze Beschreibung
BC2C	SCR INK ENCODE	Konvertiere eine Tintennummer in ein Byte, das, in den Bildschirm gepoket, alle betroffenen Punkte in dieser Tinte darstellt.
BC2F	SCR INK DECODE	Bestimme die Tintennummer des ersten Punktes von links im übergebenen Byte.
BC32	SCR SET INK	Lege die beiden Farben fest, in der eine Tinte in den beiden Blinkperioden dargestellt werden soll.
BC35	SCR GET INK	Erfrage die Farben, in der eine Tinte in den beiden Blinkperioden dargestellt wird.
BC38	SCR SET BORDER	Bestimme die Farben, in denen der Bildschirmrand in den beiden Blink-Perioden dargestellt wird.
BC3B	SCR GET BORDER	Erfrage die Farben, in denen der Bildschirmrand in den beiden Blinkperioden dargestellt wird.
BC3E	SCR SET FLASHING	Lege die Länge der beiden Blinkperioden neu fest.
BC41	SCR GET FLASHING	Erfrage, welche Länge die beiden Blinkperioden momentan haben.
BC44	SCR FILL BOX	Fülle einen rechteckigen Bildschirmausschnitt mit einer Tinte. Die Grenzen werden in Buchstabenpositionen angegeben.
BC47	SCR FLOOD BOX	Fülle einen rechteckigen Bildschirmausschnitt mit einer Tinte. Die Grenzen werden in Byte-Positionen angegeben.
BC4A	SCR CHAR INVERT	Invertiere eine Buchstabenposition (→ Cursor).

Adresse	Bezeichnung	kurze Beschreibung
BC4D	SCR HW ROLL	Scrolle den ganzen Bildschirm Hardware-mäßig um eine Buchstabenposition rauf oder runter.
BC50	SCR SW ROLL	Scrolle einen Bildschirmausschnitt um eine Buchstabenposition rauf oder runter.
BC53	SCR UNPACK	Expandiere eine Zeichen-Matrix entsprechend dem momentanen Bildschirmmodus.
BC56	SCR REPACK	Komprimiere die expandierten Bytes wieder zu einer Zeichenmatrix.
BC59	SCR ACCESS	Setze den Zeichenmodus für die Grafik-VDU.
BC5C	SCR PIXELS	Setze einen Punkt im Bildschirm.
BC5F	SCR HORIZONTAL	Zeichne eine waagerechte Linie.
BC62	SCR VERTICAL	Zeichne eine senkrechte Linie.
BD55	SCR SET POSITION	Lege die Lage des Bildschirms nur für die Software neu fest (nur CPC 664 und 6128).

THE CASSETTE MANAGER (CAS)
DIE KASSETTEN-(UND DISKETTEN-) ROUTINEN

(von AMSDOS gepatchte Routinen sind mit * markiert)

Adresse	Bezeichnung	kurze Beschreibung
BC65	CAS INITIALISE	Initialisiere die Kassetten-Routinen.
BC68	CAS SET SPEED	Lege die Schreibgeschwindigkeit neu fest.

Adresse	Bezeichnung	kurze Beschreibung
BC6B	CAS NOISY	Lege fest, ob die Kassetten-Routinen ihre Meldungen unterdrücken sollen.
BC6E	CAS START MOTOR	Starte den Motor des Kassettenrecorders.
BC71	CAS STOP MOTOR	Halte den Motor des Kassettenrecorders an.
BC74	CAS RESTORE MOTOR	Starte oder stoppe den Motor entsprechend einer früheren Einstellung.
BC77 *	CAS IN OPEN	Eröffne eine Datei zur Eingabe.
BC7A *	CAS IN CLOSE	Schließe eine Eingabedatei.
BC7D *	CAS IN ABANDON	Vergiß, daß eine Datei zur Eingabe eröffnet ist.
BC80 *	CAS IN CHAR	Lies ein Zeichen aus der Eingabedatei.
BC83 *	CAS IN DIRECT	Lies die Eingabedatei in einem Zug.
BC86 *	CAS RETURN	Gib das zuletzt gelesene Zeichen noch einmal zurück.
BC89 *	CAS TEST EOF	Frage an, ob das Ende der Eingabedatei erreicht ist.
BC8C *	CAS OUT OPEN	Eröffne eine Datei zur Ausgabe.
BC8F *	CAS OUT CLOSE	Schließe die Ausgabedatei.
BC92 *	CAS OUT ABANDON	Vergesse die Ausgabedatei. Hierbei gehen immer größere Datenmengen verloren.
BC95 *	CAS OUT CHAR	Schreibe ein Zeichen in die Ausgabedatei.
BC98 *	CAS OUT DIRECT	Schreibe die Ausgabedatei in einem Zug.

Adresse	Bezeichnung	kurze Beschreibung
BC9B *	CAS CATALOG	Erstelle ein Inhaltsverzeichnis.
BC9E	CAS WRITE	Schreibe einen Speicherbereich auf die Kassette.
BCA1	CAS READ	Lese einen Speicherbereich von Kassette ein.
BCA4	CAS CHECK	Vergleiche die Daten auf der Kassette mit einem Speicherbereich.

THE SOUND MANAGER (SOUND)
DIE LAUTSPRECHER-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
BCA7	SOUND RESET	Setze die Lautsprecher-Routinen zurück.
BCAA	SOUND QUEUE	Schicke einen Ton zum Sound-Manager.
BCAD	SOUND CHECK	Frage an, ob in der Ton-Warteschlange ein Platz für einen neuen Ton frei ist.
BCB0	SOUND ARM EVENT	Bestimme eine Routine, die aufgerufen werden soll, sobald in der Ton-Warteschlange ein Platz frei ist.
BCB3	SOUND RELEASE	Gib die Tonausgabe frei.
BCB6	SOUND HOLD	Friere die Tonausgabe ein.
BCB9	SOUND CONTINUE	Setze die Tonausgabe fort.
BCBC	SOUND AMPL ENVELOPE	Lege eine Amplituden-Hüllkurve neu fest.

Adresse	Bezeichnung	kurze Beschreibung
BCBF	SOUND TONE ENVELOPE	Lege eine Frequenz-Hüllkurve neu fest.
BCC2	SOUND A ADDRESS	Erfrage die Adresse einer Amplituden-Hüllkurve.
BCC5	SOUND T ADDRESS	Erfrage die Adresse einer Frequenz-Hüllkurve.

THE KERNEL (KL) – DIE ZENTRALE

Adresse	Bezeichnung	kurze Beschreibung
BCC8	KL CHOKE OFF	Setze den Kernel zurück.
BCCB	KL ROM WALK	Suche und initialisiere alle Hintergrund-ROMs.
BCCE	KL INIT BACK	Initialisiere ein bestimmtes Hintergrund-ROM.
BCD1	KL LOG EXT	Mache dem Kernel eine RSX-Erweiterung bekannt.
BCD4	KL FIND COMAND	Erfrage zum angegebenen Namen einer RSX oder eines ROMs die Ausführungsadresse und Speicherkonfiguration.
BCD7	KL NEW FRAME FLY	Initialisiere einen Datenblock, und füge ihn in die Liste aller Software-Unterbrechungen bei jedem Strahlrücklauf des Bildschirms ein.
BCDA	KL ADD FRAME FLY	Füge einen fertigen Datenblock in die Liste aller Software-Unterbrechungen bei jedem Strahlrücklauf ein.
BCDD	KL DEL FRAME FLY	Entferne einen Datenblock aus dieser Liste.

Adresse	Bezeichnung	kurze Beschreibung
BCE0	KL NEW FAST TICKER	Initialisiere einen Datenblock, und füge ihn in die Liste aller Software-Unterbrechungen bei jedem Interrupt ein.
BCE3	KL ADD FAST TICKER	Füge einen fertigen Datenblock in die Liste aller Software-Unterbrechungen bei jedem Interrupt ein.
BCE6	KL DEL FAST TICKER	Entferne einen Datenblock aus dieser Liste.
BCE9	KL ADD TICKER	Füge einen Datenblock in die Liste des Universal-Timers ein.
BCEC	KL DEL TICKER	Entferne einen Datenblock aus der Liste des Universal-Timers.
BCEF	KL INIT EVENT	Beschreibe einen Event-Block vorschriftsmäßig mit den in Registern angegebenen Werten.
BCF2	KL EVENT	Stoße einen Event-Block an. Dies führt zu einer Software-Unterbrechung.
BCF5	KL SYNC RESET	Lösche die Liste aller synchronisierbaren Unterbrechungen, die noch auf ihre Ausführung warten.
BCF8	KL DEL SYNCHRONOUS	Stelle den Event-Block einer synchronisierbaren Unterbrechung ruhig, und entferne ihn auch aus der Warteliste, falls er dort eingetragen ist.
BCFB	KL NEXT SYNC	Hole die nächste synchronisierbare Unterbrechung aus der Warteschlange, falls noch eine wartet.
BCFE	KL DO SYNC	Rufe die mit &BCFB geholte Unterbrechung auf.

Adresse	Bezeichnung	kurze Beschreibung
BD01	KL DONE SYNC	Beende die Bearbeitung einer synchronisierbaren Unterbrechung.
BD04	KL EVENT DISABLE	Lege die Ausführung von normalen synchronisierbaren Unterbrechungen auf Eis.
BD07	KL EVENT ENABLE	Lasse die Ausführung von normalen synchronisierbaren Unterbrechungen wieder zu.
BD0A	KL DISARM EVENT	Stelle einen Event-Block ruhig.
BD0D	KL TIME PLEASE	Erfrage den Stand des Interrupt-Zählers.
BD10	KL TIME SET	Stelle den Interrupt-Zähler auf einen neuen Startwert.
BD5B	KL RAM SELECT	Wähle eine neue RAM-Konfiguration aus (nur CPC 6128).

THE MACHINE PACK (MC) – MASCHINENNAHE ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
BD13	MC BOOT PROGRAM	Lade und starte ein Maschinencode-Programm.
BD16	MC START PROGRAM	Rufe ein Vordergrund-ROM auf.
BD19	MC WAIT FLYBACK	Warte bis zum nächsten Strahlrücklauf des Monitorbildes.
BD1C	MC SET MODE	Lege den Bildschirmmodus neu fest.
BD1F	MC SCREEN OFFSET	Setze den Hardware-Scroll-Offset.

Adresse	Bezeichnung	kurze Beschreibung
BD22	MC CLEAR INKS	Setze alle Tinten auf eine Farbe.
BD25	MC SET INKS	Lege die Farben für alle Tinten neu fest.
BD28	MC RESET PRINTER	Setze die Printer-Indirections zurück.
BD2B	MC PRINT CHAR	Versuche, ein Zeichen zum Drukker zu schicken.
BD2E	MC BUSY PRINTER	Schaue nach, ob der Drucker bereit ist.
BD31	MC SEND PRINT	Sende ein Zeichen zum Drucker.
BD34	MC SOUND REGISTER	Lade ein Byte in ein Register des Sound-Chips.
BD58	MC PRINT TRANSLATION	Ändere die Zeichen-Übersetzungs-Tabelle für den Drucker (nur CPC 664 und 6128).

JUMPER (JUMP) – EINE ROUTINE FÜR DIE SPRUNGLEISTE

Adresse	Bezeichnung	kurze Beschreibung
BD37	JUMP RESTORE	Stelle die Original-Sprungleiste wieder her.

Die Indirections der Firmware Packs

Im Gegensatz zu den Vektoren in den vorhergehenden Tabellen dienen die *Indirections* dazu, das Verhalten des Betriebssystems gezielt zu verändern. Während die Vektoren vom Betriebssystem selbst nicht angesprungen werden, ist genau das bei den Indirections der Fall. Aus diesem Grund sind die hier eingetragenen Sprünge auch nicht mit einem Restart gebildet, sondern mit dem normalen Z80-JP-Befehl.

Die Indirections werden nicht mit &BD37 JUMP RESTORE eingerichtet, sondern von dem jeweiligen Firmware Pack selbst, wenn dessen RESET- oder Initialisierungs-Routine aufgerufen wird.

Adresse	Bezeichnung	kurze Beschreibung
BDCD	IND TXT DRAW CURSOR	Zeichne einen Cursor-Fleck, falls dieser auf beiden Ebenen eingeschaltet ist.
BDD0	IND TXT UNDRAW CURSOR	Entferne den Cursor-Fleck, falls dieser auf beiden Ebenen eingeschaltet ist.
BDD3	IND TXT WRITE CHAR	Schreibe ein Zeichen auf den Bildschirm.
BDD6	IND TXT UNWRITE	Lies ein Zeichen vom Bildschirm.
BDD9	IND TXT OUT ACTION	Schreibe ein Zeichen, oder befolge einen Controlcode.
BDDC	IND GRA PLOT	Zeichne einen Punkt.
BDDF	IND GRA TEST	Bestimme die Tinte eines Punktes.
BDE2	IND GRA LINE	Zeichne eine Linie.
BDE5	IND SCR READ	Lies einen Punkt vom Bildschirm.
BDE8	IND SCR WRITE	Setze einen oder mehrere Punkte im Bildschirm unter Berücksichtigung des Grafikmodus.
BDEB	IND SCR MODE CLEAR	Lösche den Bildschirm mit Tinte 0.
BDEE	IND KM TEST BREAK	Führe den Test auf BREAK und RESET durch.

Adresse	Bezeichnung	kurze Beschreibung
BDF1	IND MC WAIT PRINTER	Versuche, ein Zeichen zum Drucker zu schicken.
BDF4	IND KM SCAN KEYS	Frage die Tastatur ab (nur CPC 664 und 6128).

THE HIGH KERNEL JUMPBLOCK
Die obere Sprungleiste des Kernel

Sowohl der obere, als auch der untere Jumpblock des *Kernel* werden ebenfalls nicht von &BD37 JUMP RESTORE eingerichtet, sondern jeweils bei der Initialisierung des gesamten Rechners: entweder beim Einschalten, CTRL/SHIFT/ESC oder beim Start eines Vordergrund-Programms: |BASIC, |CPM oder RUN "mcodename".

Der nun folgende obere Jumpblock ist aus dem ROM hierher ins RAM kopiert worden und enthält alle wesentlichen Routinen, die unabhängig vom ROM-Status lauffähig sein müssen: Interrupt, Bankumschaltung und Event-Behandlung.

Adresse	Bezeichnung	kurze Beschreibung
B900	KL U ROM ENABLE	Blende das momentan selektierte obere ROM ein.
B903	KL U ROM DISABLE	Blende das obere ROM aus und RAM ein.
B906	KL L ROM ENABLE	Blende unten das Betriebssystem-ROM ein.
B909	KL L ROM DISABLE	Blende das untere ROM wieder aus und RAM ein.
B90C	KL ROM RESTORE	Stelle einen früheren ROM-Status wieder her.
B90F	KL ROM SELECT	Wähle ein bestimmtes ROM an, und blende es ein.

Adresse	Bezeichnung	kurze Beschreibung
B912	KL CURR SELECTION	Frage an, welches ROM oben gerade selektiert ist.
B915	KL PROBE ROM	Bestimme Klasse und Version eines ROMs.
B918	KL ROM DESELECTION	Stelle eine frühere ROM-Selektion wieder her.
B91B	KL LDIR	Führe ein LDIR im RAM durch.
B91E	KL LDDR	Führe ein LDDR im RAM durch.
B921	KL POLL SYNCHRONOUS	Teste, ob eine synchronisierbare Unterbrechung auf ihre Ausführung wartet.
B92A	KL SCAN NEEDED	Teile dem Kernel mit, daß die Tastatur mit dem nächsten Interrupt abgefragt werden muß (nur CPC 664 und 6128).

THE LOW KERNEL JUMPBLOCK
Die untere Sprungleiste des Kernel

Der untere Jumpblock des KERNEL liegt an den angegebenen Adressen im ROM. Bei der Initialisierung des Rechners wird er aber auch an die entsprechenden Stellen im RAM kopiert. Er enthält einige sehr kurze Routinen, die unabhängig von der aktuellen ROM-Konfiguration verfügbar sein sollen, und ansonsten nur Sprünge in den oberen Jumpblock, wo die längeren Routinen dann fortgeführt werden.

Adresse	Bezeichnung	kurze Beschreibung
0000	RESET ENTRY RST 0	Kaltstart, totaler RESET.
0008	LOW JUMP RST 1	Sprung zu einer Routine im unteren ROM oder RAM mit Angabe

Adresse	Bezeichnung	kurze Beschreibung
0008	LOW JUMP RST 1	der gewünschten ROM-Konfiguration und Adresse in einem nachgestellten DEFW xxxx.
000B	KL LOW PCHL	Sprung zu einer Routine im unteren ROM oder RAM mit Angabe der gewünschten ROM-Konfiguration und Adresse im HL-Register.
000E	PCBC INSTRUCTION	JP (BC)
0010	SIDE CALL RST 2	Aufruf einer Routine in einem benachbarten oberen ROM mit Angabe der "Distanz" und Adresse in einem nachgestellten DEFW xxxx.
0013	KL SIDE PCHL	Aufruf einer Routine in einem benachbarten oberen ROM mit Angabe der "Distanz" und Adresse im HL-Register.
0016	PCDE INSTRUCTION	JP (DE)
0018	FAR CALL RST 3	Aufruf einer Routine im RAM oder jedem beliebigen ROM. ROM-Selektion und Adresse werden indirekt über ein nachgestelltes DEFW xxxx angezeigt.
001B	KL FAR PCHL	Aufruf einer Routine in RAM oder jedem beliebigen ROM. Die Register C und HL enthalten die gewünschte ROM-Selektion und Adresse.
001E	PCHL INSTRUCTION	JP (HL)
0020	RAM LAM RST 4	LD A,(HL) aus dem RAM.

Adresse	Bezeichnung	kurze Beschreibung
0023	KL FAR ICALL	Aufruf einer Routine im RAM oder jedem beliebigen ROM. ROM-Selektion und Adresse werden vom HL-Register angezeigt.
0028	FIRM JUMP RST 5	Sprung zu einer Routine im unteren ROM. Die gewünschte Adresse wird in einem DEFW xxxx angehängt.
0030	USER RESTART RST 6	Vom Betriebssystem nicht benutzt. Der RST 6 vom ROM speichert jedoch den momentanen RAM- und ROM-Status in Adresse &28, blendet RAM ein und springt den RST 6 im RAM an.
0038	INTERRUPT ENTRY RST 7	Der Z80 wird im Interrupt-Modus 1 betrieben. Das heißt, daß er mit jeder Interrupt-Anforderung einen RST 7 ausführt.
003B	EXT INTERRUPT	Erkennt der Kernel ein Interrupt-Signal von einer Erweiterung am Systembus, so wird diese Adresse angesprungen.

Die BASIC-Vektoren

DER EDITOR

Adresse	Bezeichnung	kurze Beschreibung
BD3A BD5B BD5E	EDIT	Editieren bzw. Neu-Eingabe eines Strings (Zeichenkette).

DIE FLIESSKOMMA-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
Zufallsgenerator:		
BD97 BDB8 BDBB	FLO RANDOMIZE 0	
BD9A BDBB BDBE	FLO RANDOMIZE	LW(HL) XOR &89656C07 → LW (SEED)
BD9D BD7C BD7F	FLO RND	RND → FLO (HL)
BDA0 BD88 BD8B	FLO LAST RND	letzter RND-Wert → FLO (HL)
Operationen:		
BD58 BD79 BD7C	FLO ADD	FLO (HL) + FLO (DE) → FLO (HL)
BD5E BD7F BD82	FLO SUB*	FLO (DE) – FLO (HL) → FLO (HL)
BD5B 349A 349A	FLO SUB	FLO (HL) – FLO (DE) → FLO (HL)
BD61 BD82 BD85	FLO MULT	FLO (HL) * FLO (DE) → FLO (HL)
BD64 BD85 BD88	FLO DIV	FLO (HL) / FLO (DE) → FLO (HL)
BD7C BD9D BDA0	FLO POT	FLO (HL) ^ FLO (DE) → FLO (HL)
BD6A BD8B BD8E	FLO VGL	SGN (FLO(HL) – FLO (DE)) → A
Funktionen:		
BD6D BD8E BD91	FLO VZW	– 1* FLO (HL) → FLO (HL)
BD79 BD9A BD9D	FLO SQR	SQR (FLO (HL)) → FLO (HL)
BD7F BDA0 BDA3	FLO LOG NAT	LOG (FLO (HL)) → FLO (HL)
BD82 BDA3 BDA6	FLO LOG DEC	LOG10 (FLO (HL)) → FLO (HL)
BD85 BDA6 BDA9	FLO POT E	E ^ FLO (HL) → FLO (HL)
BD88 BDA9 BDAC	FLO SIN	SIN (FLO (HL)) → FLO (HL)
BD8B BDAC BDAF	FLO COS	COS (FLO (HL)) → FLO (HL)
BD8E BDAF BDB2	FLO TAN	TAN (FLO (HL)) → FLO (HL)
BD91 BDB2 BDB5	FLO ARC TAN	ARCTAN (FLO (HL)) → FLO (HL)
BD55 BD76 BD79	FLO 10^A	FLO (HL) * 10^A → FLO (HL)
BD67 —— ——	FLO 2^A	FLO (HL) * 2^A → FLO (HL)
BD70 BD91 BD94	FLO SGN	SGN 8FLO(HL)) → A
Sonstiges:		
BD3D BD5E BD61	FLO MOVE	FLO (DE) → FLO (HL) und A:=exp.byte
BD76 BD97 BD9A	FLO PI	PI → FLO (HL)
BD73 BD94 BD97	FLO DEG/RAD	A=→ RAD A>0 → DEG

DIE INTEGER-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
Operatoren mit Vorzeichen (Komplementdarstellung):		
BDAC DD4F DDAF	INT ADD VZ	HL + DE → HL
BDAF DD58 DD53	INT SUB VZ	HL – DE → HL
BDB2 DD57 DD52	INT SUB* VZ	DE – HL → HL
BD5B DD60 DD5B	INT MULT VZ	HL * DE → HL
BDB8 DDA1 DD9C	INT DIV VZ	HL / DE → HL rest DE
BDBB DDA8 DDA3	INT MOD VZ	HL / DE → DE rest HL
BDC4 DE07 DE02	INT VGL	SGN (HL – DE) → A
Funktionen mit Vorzeichen:		
BDC7 DDF2 DDED	INT VZW	–1 * HL → HL
BDCA DDFE DDF9	INT SGN	SGN (HL) → A
Operationen ohne Vorzeichen:		
BDBE DD77 DD72	INT MULT	HL * DE → HL
BDC1 DDB0 DDAB	INT DIV	HL / DE → HL rest DE

KONVERTIERUNGS-ROUTINEN

Adresse	Bezeichnung	kurze Beschreibung
Konvertierung:		
BD46 BD67 BD6A	ROUND FLO TO HLA	ROUND (FLO(HL)) → HL, A=VZ
BD40 BD61 BD64	KONV HLA TO FLO	HL, A=VZ → FLO (DE)
BD43 BD64 BD67	KONV LW TO FLO	LW (HL), A=VZ → FLO (HL)
BD49 BD6A BD6D	ROUND FLO TO LW	ROUND (FLO (HL)) → LW (HL), B=VZ
BD4C BD6D BD70	FIX FLO TO LW	FIX (FLO (HL)) → LW (HL), B=VZ
BD4F BD70 BD73	INT FLO TO LW	INT (FLO (HL)) → LW (HL), B=VZ
BD94 BDB5 BDB8	KONV LW+C TO FLO	LW (HL) * 256+C → FLO (HL)

Adresse	Bezeichnung	kurze Beschreibung
BDA9 DD3C DD37	KONV HLB TO INT	HL, B0VZ → HL

Parameter für Dezimalwandlung bestimmen:

BD52 BD73 BD76	FLO PREPARE	FLO (HL) → Parameter
BDA3 DD2F DD2A	INT PREPARE VZ	HL (Integer mit VZ) → Parameter
BDA6 DD35 DD30	INT PREPARE	HL (Integer ohne VZ) → Parameter

ZUSÄTZLICHE VEKTOREN IM SCHNEIDER CPC 664 UND 6128

Adresse	Bezeichnung	kurze Beschreibung
BD3A	KM SET LOCKS	Lege den SHIFT-und den CAPS/LOCK-Status neu fest.
BD3D	KM FLUSH	Lösche alle bisher aufgelaufenen Zeichen im Tastaturpuffer.
BD40	TXT ASK STATE	Erfrage den Cursor- und VDU-Status des aktuellen Textfensters.
BD43	GRA DEFAULT	Stelle die Standardwerte für die verschiedenen Optionen der Grafik-VDU ein.
BD46	GRA SET BACK	Setze den Hintergrund-Modus der Grafik-VDU.
BD49	GRA SET FIRST	Setze die Erster-Punkt-Option für die zu zeichnenden Linien.
BD4C	GRA SET LINE MASK	Lege die Punktmaske für Linien neu fest.
BD4F	GRA FROM USER	Konvertiere die User-Koordinaten (relativ zum Origin) in Basiskoordinaten (relativ zur linken unteren Ecke).
BD52	GRA FILL	Male eine beliebige Fläche aus.
BD55	SCR SET POSITION	Lege die Lage des Bildschirms nur für die Software neu fest.
BD58	MC PRINT TRANSLATION	Ändere die Zeichen-Übersetzungstabelle für den Drucker.
B92A	KL SCAN NEEDED	Teile dem Kernel mit, daß die Tastatur mit dem nächsten Interrupt abgefragt werden muß.

ZUSÄTZLICHE INDIRCTIONS IM SCHNEIDER CPC 664 UND 6128

Adresse	Bezeichnung	kurze Beschreibung
BDF4	IND KM SCAN KEYS	Frage die Tastatur ab.

Z USÄTZLICHE VEKTOREN IM SCHNEIDER CPC 6128

Adresse	Bezeichnung	kurze Beschreibung
BD5B	KL RAM SELECT	Wähle eine neue RAM-Konfiguration aus.

THE KEY MANAGER (KM)
Die Tastatur-Routinen

BB00 KM INITIALIZE Initialisiere die Tastatur-Routinen

Eingaben: keine
Ausgaben: keine
Unverändert: IX,IY

Komplette Initialisierung des Key-Manager: Alle Variablen, Puffer und Indirections werden initialisiert.

Betroffen sind: IND KM TEST BREAK
Tastenpuffer
Expansions-Puffer und -Strings
Tasten-Übersetzungstabellen
Repeat-Verzögerung und -Geschwindigkeit
SHIFT und CAPS LOCK
BREAKs werden ignoriert

BB03 KM RESET Setze die Tastatur-Routinen zurück.

Eingaben: keine
Ausgaben: keine
Unverändert: IX,IY

"Kleine" Initialisierung des Key Managers: Die Puffer und Indirections werden initialisiert.

Betroffen sind: IND KM TEST BREAK
Tastaturpuffer
Expansions-Puffer und -Strings
BREAKs werden ignoriert

BB06 KM WAIT CHAR Warte auf ein (expandiertes) Zeichen von der Tastatur.

Eingaben: keine
Ausgaben: CY=1 und A=Zeichen
Unverändert: BC, DE, HL, IX, IY

Warte so lange, bis ein Zeichen von der Tastatur verfügbar ist. Erweiterungszeichen werden ausgewertet, zurückgegebene Zeichen ebenfalls.

BB09 KM READ CHAR Hole ein (expandiertes) Zeichen von der Tastatur (falls eins vorhanden ist).

Eingaben: keine
Ausgaben: wenn CY=1, dann A=Zeichen, sonst CY=0
Unverändert: BC, DE, HL, IX, IY

Teste, ob ein Zeichen von der Tastatur verfügbar ist (-> CY). Wenn ja, hole es. Die Routine wartet nicht. Erweiterungszeichen werden ausgewertet, ein zurückgegebenes Zeichen ebenfalls.

BB0C KM CHAR RETURN Gib ein Zeichen an den KM zurück.

Eingaben: A=Zeichen
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Man kann dem Key Manager ein einziges Zeichen zurückgeben. Beim nächsten Versuch, ein expandiertes Zeichen von der Tastatur abzuholen, wird dann dieses Zeichen ausgegeben. Es wird dabei nie expandiert, auch wenn es ein Erweiterungszeichen ist. Außerdem werden die Steuerzeichen des KM nicht befolgt (252, 253, 254). 255 kann überhaupt nicht zurückgegeben werden.

BB0F KM SET EXPAND Ordne einem Erweiterungszeichen einen String zu.

Eingaben: B = Erweiterungszeichen
C = Länge des Strings
HL = Adresse des Strings
Ausgaben: CY=1 → alles o.k.
Unverändert: IX, IY

Die Erweiterungszeichen 128 bis 159 können mit Zeichenketten (Strings) belegt werden. Dafür müssen die Strings in einem Puffer gespeichert werden.

Das String darf überall im RAM liegen, aber nicht in einem ROM.

Ein Fehler (CY=0) tritt dann auf, wenn im Puffer kein Platz mehr ist oder B keinen gültigen Code enthält.

BB12 KM GET EXPAND Hole ein Zeichen aus einem Erweiterungs-String.

Eingaben: A = Erweiterungszeichen
L = Zeichennummer im Erweiterungs-String
Ausgaben: wenn CY=1, dann A=Zeichen
Unverändert: BC, HL, IX, IY

Die Zeichen sind, beginnend mit 0, durchnumeriert.

Ein Fehler (CY=0) tritt auf, wenn A keinen gültigen Code enthält oder der String kürzer als durch L verlangt ist.

BB15 KM EXP BUFFER Lege den Speicherbereich fest, in dem der KM die Erweiterungs-Strings speichern kann.

Eingaben: DE = Adresse für den Puffer
HL = Länge
Ausgaben: CY=1 → alles o.k.
Unverändert: IX, IY

Der Puffer wird entsprechend DE und HL übernommen und mit den Standard-Expansionstrings initialisiert. Ein Fehler (CY=0) tritt auf, wenn der Puffer dafür zu kurz ist. Dann wird der alte Puffer nicht freigegeben! Der neue

Puffer muß deshalb mindestens 44 Zeichen lang sein. Der Puffer darf nur im zentralen RAM liegen.

BB18 KM WAIT KEY Warte auf ein (nicht expandiertes) Zeichen von der Tastatur.

Eingaben: keine
Ausgaben: CY=1 und A=Zeichen
Unverändert: BC, DE, HL, IX, IY

Die Routine wartet so lange, bis ein Zeichen von der Tastatur verfügbar ist. Erweiterungszeichen werden dabei nicht expandiert und zurückgegebene Zeichen nicht berücksichtigt.

BB1B KM READ KEY Hole ein (nicht expandiertes) Zeichen von der Tastatur.

Eingaben: keine
Ausgaben: wenn CY=1 dann A=Zeichen
Unverändert: BC, DE, HL, IX, IY

Hole nur dann ein Zeichen von der Tastatur, wenn eins verfügbar ist (CY=1). Diese Routine wartet nicht. Erweiterungszeichen werden nicht expandiert und zurückgegebene Zeichen nicht berücksichtigt.

BB1E KM TEST KEY Teste, ob eine bestimmte Taste gedrückt ist.

Eingaben: A = Tastennummer
Ausgaben: Z=0 → Taste ist gedrückt, sonst Z=1
CY=0 und C = Zustand von SHIFT und CTRL
Unverändert: B, DE, IX, IY

Bit 7,C = 1 → CTRL ist gedrückt
Bit 5,C = 1 → SHIFT ist gedrückt

BB21 KM GET STATE Erfrage den Status der SHIFT- und CAPS LOCK-Taste.

Eingaben: keine
Ausgaben: L = SHIFT/LOCK-Status

Ausgaben: H = CAPS/LOCK-Status
Unverändert: BC, DE, IX, IY

&00 bedeutet, daß das entsprechende LOCK nicht eingeschaltet ist.
&FF bedeutet, daß es eingeschaltet ist.

BB24 KM GET JOYSTICK Teste den Schaltzustand der Joysticks.

Eingaben: keine
Ausgaben: A und H enthalten den Status von Joystick 0
L enthält den Status von Joystick 1
Unverändert: BC, DE, IX, IY

Die einzelnen Bits der Status-Bytes bedeuten, wenn sie gesetzt (1) sind, daß folgende Taste gedrückt ist:

0 – hoch
1 – runter
2 – links
3 – rechts
4 – Feuer 1
5 – Feuer 2
6 – Pin 5 des Joystick-Anschlußes (nicht belegt)
7 – immer 0

BB27 KM SET TRANSLATE Lege das Zeichen fest, das eine Taste erzeugen soll, wenn sie ohne CTRL oder SHIFT gedrückt wird.

Eingaben: A = Tastennummer
B = neues Zeichen
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Wenn die Tastennummer größer oder gleich 80 ist, wird nichts unternommen.

Folgende Zeichen werden vom KM nicht weitergegeben, wenn die Tastatur abgefragt wird:

&FD = 253 → CAPS LOCK-Schalter
&FE = 254 → SHIFT LOCK-Schalter
&FF = 255 → Ignorierzeichen

BB2A KM GET TRANSLATE Erfrage die Belegung einer Taste ohne CTRL und SHIFT.

Eingaben: A = Tastennummer
Ausgaben: A = Zeichen, das durch diese Taste erzeugt wird
Unverändert: BC, DE, IX, IY

Siehe bei BB27 KM SET TRANSLATE wegen Zeichencodes, die der KM nicht weitergibt.

BB2D KM SET SHIFT Bestimme das Zeichen einer Taste für SHIFT.

Wie BB27 KM SET TRANSLATE, nur mit SHIFT.

BB30 KM GET SHIFT Erfrage die Tastenübersetzung mit SHIFT.

Wie BB2A KM GET TRANSLATE, nur mit SHIFT.

BB33 KM SET CONTROL Bestimme das Zeichen einer Taste für CTRL.

Wie BB27 KM SET TRANSLATE, nur mit CTRL.

BB36 KM GET CONTROL Erfrage die Tastenübersetzung mit CTRL.

Wie BB2A KM GET TRANSLATE, nur mit CTRL.

BB39 KM SET REPEAT Lege fest, ob eine Taste "Repeaten", sich also automatisch wiederholen darf, wenn man sie lange genug drückt.

Eingaben: A = Tastennummer
B=255=&FF → erlaube der Taste das Repeaten, sonst B=0
Ausgaben: keine
Unverändert: DE, IX, IY

Wenn die Tastennummer in A größer oder gleich 80 ist, wird nichts eingetragen.

BB3C KM GET REPEAT Erfrage, ob eine Taste "Repeaten" darf.

Eingaben: A = Tastennummer
Ausgaben: Z=0 → die Taste darf Repeaten,
Z=1 → sie darf nicht
Unverändert: BC, DE, IX, IY

BB3F KM SET DELAY Lege die Verzögerungszeiten beim "Repeaten" fest.

Eingaben: H = Wartezeit bis zum ersten Repeat
L = Wartezeit zwischen weiteren Repeats
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Die Verzögerungszeiten beziehen sich auf die Tastaturabfragen, und die Tastatur wird normalerweise 50mal pro Sekunde abgefragt. Die Standardwerte sind 30 (0.6 sec.) für die erste, und 2 (0.04 sec.) für die folgenden Wartezeiten.

Wenn der Tastaturpuffer nicht leer ist, werden Repeats so lange hinausgezögert, bis das laufende Programm den Puffer geleert hat. Dadurch wird verhindert, daß der Tastaturpuffer sich unbemerkt durch Repeats füllt, wenn das Programm mit der Abarbeitung nicht nachkommt.

BB42 KM GET DELEAY Erfrage die Verzögerungszeiten beim "Repeaten".

Eingaben: keine
Ausgaben: H = Wartezeit bis zum ersten Repeat
L = Wartezeit zwischen folgenden Repeats
Unverändert: BC, DE, IX, IY

Siehe auch Anmerkungen bei BB3F KM SET DELAY.

BB45 KM ARM BREAK Aktiviere den Break-Mechanismus.

Eingaben: DE = Adresse der Break-Behandlungsroutine
C = Benötigte ROM-Selekt-Adresse
Ausgaben: keine
Unverändert: IX, IY

Das BREAK-Event ist immer synchron, express mit der Priorität 0 und mit Far Address.

BB48 KM DISARM BREAK Schalte den Break-Mechanismus aus.

Eingaben: keine
Ausgaben: keine
Unverändert: BD, DE, IX, IY

BB4B KM BREAK EVENT Löse ein Break-Event aus, wenn der Mechanismus aktiv ist.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Wenn der Break-Mechanismus ausgeschaltet ist, geschieht nichts.

Sonst wird der Break-Eventblock in die Synchron Pending Queue eingehängt und das BREAK-Event-Token (&EF=139) in den Tastenpuffer eingefügt (außer, wenn dieser voll ist) und der Break-Mechanismus wieder ausgeschaltet.

Diese Routine ist dafür vorgesehen, vom Interrupt-Pfad aus aufgerufen zu werden. Sie läßt also keine Interrupts zu, wohl aber der Vektor BB4B, da dieser ja mit einem LOW JUMP gebildet ist. Man muß sich die eigentliche Routinenadresse aus dem Vektor herausholen und die Routine direkt aufrufen.

BD3A KM SET LOCKS Lege den SHIFT- und den CAPS LOCK-Status neu fest (nur CPC 664 und 6128).

Eingaben: H = neuer CAPS LOCK- Status
L = neuer SHIFT LOCK-Status
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Der Zustand von CAPS LOCK (Großbuchstaben-Arretierung) und SHIFT LOCK (Arretierung der SHIFT-Taste für alle Zeichen) wird entsprechend H und L neu gesetzt. Dabei gilt folgende Vereinbarung:

&00 → ausschalten
&FF → einschalten

BD3D KM FLUSH Lösche alle bisher aufgelaufenen Zeichen im Tastaturpuffer (nur CPC 664 und 6128).

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Der Tastaturpuffer wird komplett gelöscht, ein eventuell angefangenes Erweiterungszeichen oder ein Put Back Character werden vergessen.

Besitzer des CPC 464 können als Ersatz folgende Routinen benutzen:

BASIC:
```
WHILE INKEY$<>"":WEND
```

ASSEMBLER:
```
LOOP12: CALL #BB09              ; KM READ CHAR
        JR   NC,LOOP12
        RET
```

THE TEXT VDU (TXT) – Die Textausgabe-Routinen

BB4E TXT INITIALISE Initialisiere die Text-VDU.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Komplette Initialisierung der Text-VDU.

Betroffen sind: die Indirections der Text-VDU
die Controlcode-Funktionstabelle
Zeichensatz (alle 256 Zeichen aus dem ROM, keine aus dem RAM) Text-Stream 0 wird angewählt

Alle Streams werden normal eingestellt: PAPER 0, PEN 1, Window = ganzer Screen, Cursor = enabled & off, Hintergrund-Modus opaque (deckend), VDU enabled (CHR$(6)), TAGOFF, LOCATE 1,1

BB51 TXT RESET Setze die Text-VDU zurück.

Eingaben: keine
Ausgaben: keine

Unverändert: IX, IY

Kleine Initialisierung der Text-VDU.

Betroffen sind: die Indirections der Text-VDU
die Controlcode-Funktionstabelle

BB54 TXT VDU ENABLE Erlaube, daß Zeichen im aktiven Fenster ausgegeben werden.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Das Ausdrucken der Zeichen im aktuellen Fenster wird wieder zugelassen und der Cursor aktiviert (enabled). Der Controlcode-Puffer wird geleert; wirkt auf TXT OUTPUT und TXT WR CHAR.

BB57 TXT VDU DISABLE Verbiete die Ausgabe von Zeichen im aktiven Fenster.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Das Ausdrucken der Zeichen im aktuellen Fenster wird verboten. Der Cursor wird abgeschaltet (disabled). Der Controlcode-Puffer wird geleert.

Beim CPC 464 werden Controlcodes aber weiterhin befolgt und der Cursor auch weiterhin mit jedem Zeichen nach rechts bewegt.

Für den CPC 664 und 6128 tritt jedoch eine wichtige Änderung ein, was die Befolgung der Controlcodes betrifft (siehe dazu &BBB1 TXT GET CONTROLS).

Dieser Vektor wirkt auf &BB5A TXT OUTPUT und &BB5D TXT WR CHAR.

BB5A TXT OUTPUT Drucke ein Zeichen, oder befolge den Controlcode.

Eingaben: A = Zeichen, Controlcode oder dessen Parameter
Ausgaben: keine
Unverändert: alle Register bleiben erhalten

Wartet ein vorher ausgegebener Controlcode noch auf einen Parameter, so wird A als Parameter behandelt.

Sonst wird das Zeichen über IND TXT WR CHAR gedruckt (A >= 32) oder selbst als Controlcode interpretiert (A < 32).

Wenn der Text-Stream nicht aktiviert, also disabled, ist, wird das Zeichen nicht gedruckt.

Controlcodes werden beim CPC 464 aber trotzdem befolgt. Die Änderungen hierbei für die CPCs 664 und 6128 sind bei &BBB1 TXT GET CONTROLS beschrieben.

Ist der Grafikmodus (TAG) eingeschaltet, werden alle Zeichen (0 bis 255) über GRA WR CHAR ausgedruckt.

BB5D TXT WR CHAR Drucke ein Zeichen. Controlcodes erzeugen Sonderzeichen.

Eingaben: A = Zeichen
Ausgaben: keine
Unverändert: IX, IY

Drucke das Zeichen A im momentan aktiven Textfenster aus. Benutzt wird dazu IND TXT WRITE CHAR.

Das Zeichen wird nicht ausgedruckt, wenn das aktuelle Fenster nicht aktiviert (disabled) ist.

BB60 TXT RD CHAR Versuche, ein Zeichen vom Bildschirm zu lesen.

Eingaben: keine
Ausgaben: CY=1 → A=Zeichen, sonst CY=0 → nicht identifizierbar
Unverändert: BC, DE, HL, IX, IY

Die Text-VDU versucht, die Grafikinformation im Bildschirm auf der Cursorposition des aktuellen Textfensters in einen Zeichencode zurückzuverwandeln. Dazu wird IND TXT UNWRITE benutzt.

Der Cursor muß beim CPC 464 vorher in das Textfenster zurückgezwungen werden (siehe TXT VALIDATE), sonst werden eventuell Zeichen von außerhalb des Fensters gelesen.

Beim CPC 664 oder 6128 erfolgt das automatisch und kann dazu führen, daß das Textfenster gescrollt wird.

BB63 TXT SET GRAPHIC Bestimme, ob der Text im laufenden Fenster auf der Cursor- oder Grafikposition ausgegeben werden soll.

Eingaben: A = Schaltflag
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Ist A=0 wird der Text auf der Cursorposition ausgegeben (TAGOFF) ...
Ist A>0 wird der Text auf der Position des Grafikcursors ausgegeben (TAG).

Betroffen hiervon ist nur TXT OUTPUT.

Ist der Grafik-Schreibmodus akiviert (TAG), so werden Controlcodes nicht befolgt, sondern als Sonderzeichen gedruckt. Außerdem ist das Ausschalten der Text-VDU (TXT VDU DISABLE) unwirksam, die Zeichen werden trotzdem ausgegeben.

BB66 TXT WIN ENABLE Lege die Grenzen des aktuellen Textfensters fest.

Eingaben: H und D = linke und rechte Spalte
L und E = obere und untere Zeile des Textfensters
Ausgaben: keine
Unverändert: IX, IY

Die angegebenen Zeilen- und Spaltengrenzen werden automatisch nach ihrer Größe sortiert und entsprechend dem Bildschirm-Modus auf die maximal möglichen Werte reduziert.

Bezugspunkt ist die linke, obere Ecke mit den Koordinaten (0,0). Angegeben werden jeweils die inneren Grenzen des Fensters wie beim WINDOW-Kommando in BASIC.

BB69 TXT GET WINDOW Erfrage die Grenzen des aktuellen Textfensters.

Eingaben: keine
Ausgaben: H und D enthalten die linke und rechte Spalte des Fensters
L und E enthalten die obere und untere Zeile des Fensters
CY=0 → Fenster bedeckt den ganzen Bildschirm
Unverändert: BC, IX, IY

Angegeben werden die inneren Fenstergrenzen. Bezugspunkt ist die linke obere Ecke mit den Koordinaten (0,0).

BB6C TXT CLEAR WINDOW Lösche die Anzeige im aktuellen Textfenster.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Das aktuelle Textfenster wird mit dessen Paper-INK gelöscht. Der Cursor wird in die linke obere Ecke gesetzt.

BB6F TXT SET COLUMN Bewege den Cursor des aktuellen Textfensters in die angegebene Spalte.

Eingaben: A = neue Cursor-Spalte
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Bezugspunkt ist die linke Spalte des aktuellen Textfensters mit der Spaltennummer 1.

Als neue X-Koordinate des Cursors kann auch ein Wert außerhalb des Fensters angegeben werden. Vor Ausgabe eines Zeichens oder des Cursors wird er automatisch in das Fenster zurückgezwungen (siehe TXT VALIDATE).

BB72 TXT SET ROW Bewege den Cursor des akt. Textfensters in die angegebene Zeile.

Eingaben: A = neue Cursor-Zeile
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Bezugspunkt ist die oberste Zeile des aktuellen Textfensters mit der Zeilennummer 1.

Die neue Y-Koordinate des Cursors kann auch außerhalb des Textfensters liegen. Vor Ausgabe eines Zeichens oder des Cursors wird er automatisch ins Fenster zurückgezwungen (siehe TXT VALIDATE).

BB75 TXT SET CURSOR Lege Zeile und Spalte des Cursors neu fest.

Eingaben:	H = neue Spalte
	L = neue Zeile des Cursors
Ausgaben:	keine
Unverändert:	BC, DE, IX, IY

Bezugspunkt ist die obere linke Ecke des aktuellen Textfensters mit der Koordinate (1,1).

Die neuen Koordinaten des Cursors konnen auch außerhalb des Textfensters liegen. Vor Ausgabe eines Zeichens oder des Cursors wird er automatisch ins Fenster zurückgezwungen (siehe TXT VALIDATE).

BB78 TXT GET CURSOR Erfrage Zeile und Spalte des Cursors.

Eingaben:	keine
Ausgaben:	H = Spalte und
	L = Zeile des Cursors
	A = Scroll-Zähler
Unverändert:	BC, DE, IX, IY

Bezugspunkt ist die obere linke Ecke des aktuellen Textfensters mit der Koordinate (1,1).

Die angegebene Position muß nicht unbedingt im Textfenster liegen. Dann wird der Cursor vor Ausgabe des nächsten Zeichens oder des Cursors ins Textfenster zurückgezwungen (siehe TXT VALIDATE).

Der Scroll-Zähler wird im Betriebssystem nicht weiter benutzt, ist aber ganz nützlich, wenn man feststellen will, ob das Fenster seit einem früheren Zeitpunkt gescrollt wurde.

Mit jedem Scroll nach oben wird der Zähler vermindert, mit jedem Scroll nach unten erhöht.

BB7B TXT CUR ENABLE Schalte Cursor auf der Benutzerebene ein.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Das Ein- und Ausschalten des Cursors mittels TXT CUR ENABLE und TXT CUR DISABLE ist für das Anwenderprogramm vorgesehen. Der Cursor wird aber auch "disabled", wenn TXT VDU DISABLE benutzt wird. Diese Funktionen sind auch mit den Controlcodes 2 und 3 erreichbar.

BB7E TXT CUR DISABLE Schalte den Cursor auf der Benutzer-Ebene aus.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Siehe &BB7B TXT CUR ENABLE.

BB81 TXT CUR ON Schalte den Cursor auf der Systemebene ein.

Eingaben: keine
Ausgaben: keine
Unverändert: alle Register bleiben erhalten

Das Ein- und Ausschalten des Cursors mittels TXT CUR ON und TXT CUR OFF ist für die "tieferen Ebenen" vorgesehen, so zum Beispiel Betriebssystem, Editor oder BASIC. Letzteres hält den Cursor aber leider immer "off", so daß ein BASIC-Programm ihn nicht mit "enable" einschalten kann. In diesem Fall kann man aber ganz leicht die Vektoren aufrufen, da diese keine Ein- und Ausgabebedingungen haben.

BB84 TXT CUR OFF Schalte den Cursor auf der Systemebene aus.

Eingaben: keine
Ausgaben: keine
Unverändert: alle Register bleiben erhalten

Siehe &BB81 TXT CUR ON.

BB87 TXT VALIDATE Überprüfe, ob sich eine Position innerhalb des Textfensters befindet.

Eingaben: H = Spalte der Prüfposition
L = Zeile der Prüfposition
Ausgaben: H = Spalte der ins Fenster zurückgezwungenen Position
L = Zeile der ins Fenster zurückgezwungenen Position
CY=1 → Das Textfenster würde dadurch nicht gescrollt
CY=0 und B=255 → Das Fenster würde hochgerollt
CY=0 und B=0 → Das Fenster würde runtergerollt
Unverändert: BC, DE, IX, IY

Bezugspunkt der Koordinatenangaben ist die linke, obere Ecke mit den Koordinaten (1,1).

Bevor ein Zeichen oder der Cursor ausgegeben und bevor ein Controlcode übernommen wird, überprüft (validates) die Text-VDU die momentane Cursorposition im aktuellen Textfenster und scrollt dieses, wenn nötig. Danach liegt die Cursorposition dann sicher im Textfenster, und das Zeichen kann ausgegeben werden.

TXT VALIDATE allein scrollt noch nicht!

BB8A TXT PLACE CURSOR Male einen "Cursor-Fleck" auf die Cursorposition.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Sollen in einem Programm mehrere Cursor in einem Textfenster bereitgestellt werden, können zusätzliche Cursor-Flecken mit TXT PLACE CURSOR und TXT REMOVE CURSOR erzeugt und wieder entfernt werden. Die Kontrolle dieser Cursor-Flecken obliegt aber dem Anwenderprogramm. Beispiel für einen zusätzlichen Cursor ist der Copy-Cursor des Zeileneditors.

BB8D TXT REMOVE CURSOR Entferne einen "Cursor-Fleck" von der Cursorposition.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Siehe TXT PLACE CURSOR.

BB90 TXT SET PEN Lege die Stifttinte für die Buchstaben fest.

Eingaben: A = Tintennummer (PEN-INK)
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Die Vordergrund-Tinte für das aktuelle Textfenster wird mit A neu festgelegt. Mit dieser Tinte werden die Buchstaben selbst gezeichnet.

Die Tintennummer wird automatisch mit 1, 3 oder 15 maskiert, um eine im aktuellen Bildschirmmodus gültige Tintennummer zu erhalten.

Außerdem wird der Cursor an die neue Vordergrund-Tinte angepaßt, wenn er im aktuellen Textfenster sichtbar ist.

BB93 TXT GET PEN Erfrage die momentane Stifttinte.

Eingaben: keine
Ausgaben: A = Vordergrund-Tinte (PEN-INK)
Unverändert: BC, DE, HL, IX, IY

Die Vordergrund-Tinte ist die INK, in der im aktuellen Textfenster die Buchstaben selbst gemalt werden.

BB96 TXT SET PAPER Lege die Hintergrund-Tinte für die Buchstaben fest.

Eingaben: A = Hintergrund-Tinte (PAPER-INK)
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Die Tintennummer wird mit 1, 3 oder 15 maskiert, um eine im momentanen Bildschirm-Modus gültige Tintennummer zu erhalten.

Die Tintennummer gilt nur für das aktuelle Textfenster. Der Cursor wird automatisch an die neue PAPER-INK angepaßt, wenn er sichtbar ist.

Die Hintergrund-Tinte ist die INK, mit der das Buchstabenfeld um den eigentlichen Buchstaben herum gefüllt wird, wenn ein Zeichen gemalt wird. Außerdem werden mit dieser Farbe Teile oder das ganze Textfenster gelöscht (TXT CLEAR WINDOW und Controlcodes 12, 16, 17, 18, 19 und 20).

BB99 TXT GET PAPER Erfrage die momentane Hintergrund-Tinte.

Eingaben: keine
Ausgaben: A = Hintergrund-Tinte (PAPER-INK)
Unverändert: BC, DE, HL, IX, IY

Die Hintergrund-Tinte ist die INK, in der im aktuellen Textfenster das Buchstabenfeld um den eigentlichen Buchstaben herum gefüllt wird, wenn ein Zeichen gemalt wird.

BB9C TXT INVERSE Tausche die Papier- und Stifttinten für das momentan aktuelle Textfenster aus.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Die Vordergrund- und Hintergrund-Tinte des aktuellen Textfensters werden vertauscht.

Nur CPC 464: Der Cursor wird nicht an die neuen INKs angepaßt und sollte deshalb möglichst nicht sichtbar sein. Solange aber der "Standard-Cursor" des CPC verwendet wird, d.h., solange IND TXT DRAW CURSOR und IND TXT UNDRAW CURSOR nicht gepatcht werden, kann man darauf verzichten, da dessen primitive Cursor-Zeichenroutine in beiden Fällen den gleichen Cursor liefert.

Beim CPC 664 und 6128 wird der Cursor-Fleck aber vorher entfernt und nachher wieder neu gezeichnet.

BB9F TXT SET BACK Lege fest, ob die Buchstaben im Transparentmodus geschrieben werden sollen.

Eingaben: A = Hintergrund-Flag
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Mit A wird festgelegt, ob im aktuellen Textfenster der Hintergrund wie normal opaque, also deckend, oder ob der Hintergrund durchsichtig, also transparent sein soll.

A=0 → opaque: Beim Zeichnen eines Buchstabens wird der Hintergrund mit der PAPER-INK gelöscht.

A>0 → transparent: Beim Zeichnen der Buchstaben werden nur die Vordergrund-Punkte, also die Punkte des Buchstabens selbst gesetzt. Der Hintergrund bleibt unverändert, die alten Grafikinformationen des Bildschirms scheinen noch zwischen den Buchstaben durch.

Hiervon wird nur die Textausgabe auf der Cursorposition beeinflußt. Bei der Ausgabe von Zeichen auf der Grafikposition (TAG) wird dieses Flag ignoriert. Für CPC 664 und 6128 gibt es eine entsprechende Einstellung für die Grafikausgabe.

BBA2 TXT GET BACK Erfrage, ob im aktuellen Textfenster der Transparentmodus eingeschaltet ist.

Eingaben: keine
Ausgaben: A = Hintergrund-Flag
Unverändert: BC, IX, IY

Siehe &BB9F TXT SET BACK.

BBA5 TXT GET MATRIX Erfrage, an welcher Stelle die Grafikinformation über das Aussehen des angegebenen Buchstaben abgelegt ist.

Eingaben: A = Zeichencode
Ausgaben: HL zeigt auf die Zeichenmatrix
CY=1 → Die Matrix befindet sich im RAM
CY=0 → Die Matrix ist im unveränderbaren Bereich im ROM
Unverändert: BC, DE, IX, IY

Um ein Zeichen auf dem Bildschirm darzustellen, wird dessen Matrix in den Bildschirm übertragen. Die Matrix besteht aus 8 Bytes zu je 8 Bits.

Das erste Byte definiert die oberste Punktzeile, die Bits 7 jeweils die linke Punktspalte. Ein gesetztes Bit (Bit=1) deutet an, daß der entsprechende Punkt in der jeweils gültigen Vordergrund-Tinte gesetzt werden soll.

BBA8 TXT SET MATRIX Bestimme das Aussehen eines Buchstaben neu.

Eingaben: A = Zeichencode
HL zeigt auf die neue Matrix
Ausgaben: CY=1 → o.k.
Unverändert: IX, IY

Die Matrix für ein Zeichen ist 8 Bytes lang. Ist das Zeichen nicht redefinierbar im RAM abgelegt, unternimmt diese Routine nichts, kehrt aber mit nicht gesetztem CY-Flag zurück (CY=0).

Die neue Matrix kann auch direkt aus einem oberen ROM gesetzt werden. Dazu muß das ROM nur eingeblendet sein. Da der im Vektor verwendete Restart aber immer das obere ROM ausblendet, muß man sich die Adresse dann aus dem Vektor holen und die Routine direkt anspringen.

Siehe auch &BBA5 TXT GET MATRIX.

BBAB TXT SET M TABLE Teile der Text-VDU mit, in welchem Speicherbereich die selbstdefinierbaren Zeichenmatrizen abgelegt werden können.

Eingaben: D = Flag
E = Code des ersten Zeichens für die RAM-Tabelle
HL = Adresse der RAM-Tabelle
Ausgaben: CY=0 → Es gab vorher keine Tabelle im RAM
CY=1 → A = altes erstes Zeichen für die RAM-Tabelle und
HL = alte Adresse der RAM-Tabelle
Unverändert: IX, IY

Ist D>0 wird die alte Matrixtabelle im RAM vollständig gelöscht (SYMBOL AFTER 256).

Ist D=0, wird im RAM eine neue Tabelle entsprechend E und HL installiert. Dabei muß man darauf achten, daß man auch wirklich genug Speicher für die Tabelle zur Verfügung stellt: Der benötigte Platz ist (256–E)*8 Bytes. Die Tabelle muß vollständig im zentralen RAM liegen.

In dieser Tabelle werden die redefinierbaren Zeichen in aufsteigender Reihenfolge bis zum Zeichen 255 gespeichert.

Die neue Tabelle wird mit den bisher gültigen Zeichen aus der alten RAM-Tabelle und, falls nötig, aus dem ROM initialisiert. Überlappen sich dabei in der

alten und in der neuen RAM-Tabelle der Bereich der übernommenen Buchstaben, so darf die neue Tabelle nicht nach oben versetzt sein. Die alten Matrizen werden nämlich generell mit LDIR verschoben.

Siehe auch &BBA5 TXT GET MATRIX.

BBAE TXT GET TABLE Erfrage, wo momentan der Bereich für die selbstdefinierbaren Zeichenmatrizen liegt.

Eingaben: keine
Ausgaben: CY=0 → Es ist kein Bereich für redefinierbare Zeichen im RAM festgelegt worden
CY=1 → A = 1. Zeichen der RAM-Tabelle
HL zeigt auf die Tabelle
Unverändert: BC, DE, IX, IY

Siehe &BBA5 TXT GET MATRIX und &BBA8 TXT SET TABLE.

BBB1 TXT GET CONTROLS Erfrage die Lage der Controlcode-Tabelle.

Eingaben: keine
Ausgaben: HL zeigt auf die Controlcode-Tabelle
Unverändert: AF, BC, DE, IX, IY

Die Controlcode-Tabelle enthält zu jedem Controlcode (0 bis 31) in aufsteigender Reihenfolge in jeweils 3 Bytes folgende Informationen:

DEFB Anzahl benötigter Parameter (maximal 9)
DEFW Routinenadresse, die im zentralen RAM liegen muß.

Durch Änderung eines Eintrags in diese Tabelle kann einem Controlcode eine völlig neue Funktion zugeordnet werden.

Die Behandlungsroutinen der Controlcodes müssen folgende Ein- und Ausprungsbedingungen erfüllen:

Eingaben:
- A und C enthalten den letzten Parameter oder den Controlcode selbst, wenn dieser keine Parameter benötigt
- B = Anzahl Parameter + 1
- HL zeigt vor den ersten Parameter (dort steht der Controlcode selbst)

Ausgaben: keine

Die Register AF, BC, DE und HL dürfen verändert werden.

Die Controlcodes werden von &BB5A TXT OUTPUT auch dann befolgt, wenn die Textausgabe für den aktiven Stream ausgeschaltet ist (mit Controlcode 21 NAK oder &BB57 TXT VDU DISABLE).

Das gilt jedoch nur für den CPC 464. Ab dem CPC 664 wird in die Controlcode-Tabelle ein zusätzliches Flag eingetragen, das festlegt, ob ein Controlcode vom VDU-disable/enable-Status eines Textfensters betroffen sein soll oder nicht. Dieses Flag ist in Bit 7 des ersten Bytes jedes Tabelleneintrags unterge- bracht, also in dem Byte mit der Anzahl der Parameter. Hier gilt:

Bit 7 = 0 → Controlcode wird trotzdem befolgt
Bit 7 = 1 → Ausführung ist vom VDU-Status abhängig

Interessant ist, daß hier die Kompatibilität zum CPC 464 kaum gewahrt wurde: Praktisch alle Controlcodes werden vom VDU-Status beeinflußt! Ausgenommen sind nur CHR$(6) ACK, mit dem die Textausgabe wieder aktiviert werden kann, und CHR$(27) ESC, das sowieso ignoriert wird.

BBB4 TXT STREAM SELECT Wähle ein anderes Textfenster an.

Eingaben:	A = Nummer des neuen Textfensters
Ausgaben:	A = Nummer des alten Textfensters
Unverändert:	BC, DE, IX, IY

A wird mit 7 maskiert, um eine gültige Stream-Nummer im Bereich 0 bis 7 zu erhalten. Ist das gewünschte Textfenster bereits angewählt, kehrt die Routine sofort zurück. Es ist daher empfehlenswert, vor jeder Textausgabe prinzipiell noch einmal das gewünschte Textfenster anzuwählen.

Folgende Parameter werden für jedes Fenster getrennt verwaltet:

PEN INK:	(Vordergrund-Tinte)
PAPER INK:	(Hintergrund-Tinte)
Cursorposition:	Spalte und Zeile
Fenstergrenzen:	links, rechts, oben und unten
Cursor-Status:	ON/OFF und ENABLE/DISABLE
VDU-Status:	ENABLE/DISABLE
Hintergrund-Modus:	OPAQUE/TRANSPARENT
Grafik-Schreibmodus:	TAG/TAGOFF

BBB7 TXT SWAP STREAMS Vertausche die Einstellungen und Parameter zweier Textfenster.

Eingaben: B und C enthalten die Nummern der Textfenster
Ausgaben: keine
Unverändert: IX, IY

Beide Stream-Nummern werden mit 7 maskiert, um gültige Werte sicherzustellen. Die Nummer des aktuell angewählten Textfensters ändert sich nicht, auch wenn es einer der beiden Tauschpartner ist. Alle Parameter, die für die Streams getrennt verwaltet werden, werden ausgetauscht.

Siehe dazu &BBB4 TXT STR SELECT.

BD40 TXT ASK STATE Erfrage Cursor- und VDU-Status des aktuellen Textfensters (nur CPC 664 und 6128).

Eingaben: keine
Ausgaben: A=Status
Unverändert: BC, DE, HL, IX, IY

A enthält in insgesamt drei Bits folgende Informationen:

Bit 0 – Cursor enabled (0) oder disabled (1) (User-Ebene)
Bit 1 – Cursor on (0) oder off (1) (Systemebene)
Bit 7 – VDU enabled (0) oder disabled (1)

THE GRAPHICS VDU (GRA)
Die Grafik-Routinen

BBBA GRA INITIALISE Initialisiere die Grafik-VDU.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Die Grafik-VDU wird komplett initialisiert.

Betroffen sind: die Indirections der Grafik-VDU
Grafik-PAPER := 0 (Hintergrund-Tinte)
Grafik-PEN := 1 (Vordergrund-Tinte)

Grafik-WINDOW := ganzer Bildschirm
ORIGIN 0,0
MOVE 0,0

CPC 664 und 6128 zusätzlich:

Hintergrund-Modus := opaque (deckend)
Linienmaske := &FF (durchgezogene Linie)
Erster-Punkt := wird immer gesetzt

BBBD GRA RESET Setze die Grafik-VDU zurück.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Kleine Initialisierung der Grafik-VDU. Es werden nur die Indirections auf die Standard-Routinen eingestellt. Bei CPC 66 und 6128 werden zusätzlich auch der Grafik-Hintergrund-Modus, die Linienmaske und die Erster-Punkt-Option auf ihre Default-Werte eingestellt.

BBC0 GRA MOVE ABSOLUTE Bewege den Grafik-Cursor zur angegebenen Position

Eingaben: DE = X-Koordinate des Zielpunktes
HL = Y-Koordinate des Zielpunktes
Ausgaben: keine
Unverändert: IX, IY

Die Zielkoordinaten werden relativ zum Origin angegeben und sind vorzeichenbehaftet (–32768 bis +32767).

BBC3 GRA MOVE RELATIVE Bewege den Grafik-Cursor relativ zur momentanen Position.

Eingaben: DE = X-Versatz und
HL = Y-Versatz
Ausgaben: keine
Unverändert: IX, IY

Die Zielkoordinaten werden relativ zur aktuellen Position des Grafik-Cursors angegeben und sind vorzeichenbehaftet (–32768 bis +32767).

BBC6 GRA ASK CURSOR Erfrage die momentane X- und Y-Koordinate des Grafik-Cursors.

Eingaben: keine
Ausgaben: DE = X-Koordinate
HL = Y-Koordinate
Unverändert: BC, IX, IY

Die Koordinaten werden relativ zum Origin angegeben und sind vorzeichenbehaftet (–32768 bis +32767).

BBC9 GRA SET ORIGIN Lege den Bezugspunkt für die absolute Cursorpositionierung neu fest.

Eingaben: DE = X-Koordinate
HL = Y-Koordinate
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zur linken unteren Ecke mit den Koordinaten (0,0) und vorzeichenbehaftet (–32768 bis +32767).

BBCC GRA GET ORIGIN Erfrage den momentanen Grafik-Nullpunkt.

Eingaben: keine
Ausgaben: DE = X-Koordinate
HL = Y-Koordinate
Unverändert: AF, BC, IX, IY

Die Koordinatenangaben sind relativ zur linken unteren Ecke mit den Koordinaten (0,0) und vorzeichenbehaftet (–32768 bis +32767).

BBCF GRA WIN WIDTH Lege den linken und rechten Rand des Grafikfensters fest.

Eingaben: DE und HL enthalten die X-Koordinaten der linken und rechten Grenze
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zur linken unteren Ecke mit den Koordinaten (0,0) und vorzeichenbehaftet (–32768 bis +32767).

DE und HL bestimmen die linke und rechte innere Fenstergrenze, wobei der kleinere Wert automatisch für den linken Rand genommen wird. DE und HL werden eventuell soweit verkleinert, bis das Fenster auf den Bildschirm paßt.

Außerdem werden die Fenstergrenzen auf ganze Bytes erweitert, die linke nach links und die rechte nach rechts. Die tatsächlichen Fenstergrenzen ergeben sich (wenn DE < HL) dann als:

links = DE and &FFF8
rechts = HL or &0007

BBD2 GRA WIN HEIGHT Lege den oberen und unteren Rand des Grafik-Fensters fest.

Eingaben: DE und HL enthalten die Y-Koordinaten der oberen und unteren Fenstergrenze.
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zur linken unteren Ecke mit den Koordinaten (0,0) und vorzeichenbehaftet (–32768 bis +32767).

DE und HL geben die obere und untere innere Grenze des Grafikfensters an, wobei automatisch der kleinere Wert für die untere Grenze benutzt wird. DE und HL werden soweit verkleinert, bis das Fenster auf den Bildschirm paßt.

Außerdem werden die Koordinaten auf Rasterzeilen-Grenzen erweitert. Da eine Rasterzeile des Monitorbildes zwei logische Grafikzeilen umfaßt, ergeben sich (wenn DE < HL) die tatsächlichen Grenzen als:

oben = HL or &0001
unten = DE and &FFFE

BBD5 GRA GET W WIDTH Erfrage den linken und rechten Rand des Grafikfensters.

Eingaben: keine
Ausgaben: DE und HL = linke und rechte Fenstergrenze
Unverändert: BC, IX, IY

Die Koordinatenangaben sind relativ zur linken unteren Ecke mit den Koordinaten (0,0) und vorzeichenbehaftet (–32768 bis +32767).

Angegeben werden die inneren Grenzen, also die X-Koordinaten des ersten und des letzten Punktes innerhalb des Fensters.

BBD8 GRA GET W HEIGHT Erfrage den oberen und unteren Rand des Grafikfensters.

Eingaben: keine
Ausgaben: DE und HL enthalten die Y-Koordinate der oberen und unteren Fenstergrenze.
Unverändert: BC, IX, IY

Die Koordinatenangaben sind relativ zur linken unteren Ecke mit den Koordinaten (0,0) und vorzeichenbehaftet (–32768 bis +32767).

Es werden die inneren Fenstergrenzen angegeben, also die Y-Koordinaten der obersten und der untersten Zeile des Grafikfensters.

BBDB GRA CLEAR WINDOW Lösche die Anzeige im Grafikfenster.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Das Grafikfenster wird mit der Grafik-PAPER-INK gelöscht. Der Grafik-Cursor wird zum Origin bewegt.

BBDE GRA SET PEN Lege die Tinte des Zeichenstiftes neu fest.

Eingaben: A = Nummer der Vordergrund-Tinte (Grafik-PEN-INK)
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Die Tintennummer wird mit 1, 3 oder 15 maskiert, um einen im momentan eingestellten Bildschirm-Modus gültigen Wert zu erhalten.

Die Grafik-PEN-INK wird benutzt für PLOT, DRAW und um Zeichen an der Position des Grafik-Cursors auszugeben. Bei CPC 664 und 6128 ist sie auch eine der Tinten der für &BD52 GRAFILL.

BBE1 GRA GET PEN Erfrage die momentane Tinte des Zeichenstiftes.

Eingaben: keine
Ausgaben: A = Nummer der Vordergrund-Tinte (Grafik-PEN-INK)
Unverändert: BC, DE, HL, IX, IY

BBE4 GRA SET PAPER Lege die Hintergrund-Tinte für die Grafik-VDU neu fest.

Eingaben: A = neue Hintergrund-Tinte (Grafik-PAPER-INK)
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Die Tintennummer wird mit 1, 3 oder 15 maskiert, um einen im momentan eingestellten Bildschirmmodus gültigen Wert zu erhalten.

Die Grafik-PAPER-INK wird benutzt, um das Grafikfenster zu löschen und um den Hintergrund von Buchstaben zu zeichnen, die an der Position des Grafik-Cursors ausgegeben werden. Außerdem wird beim Testen von Punkten außerhalb des Grafikfensters immer diese Tinte angegeben. Beim CPC 664 und 6128 wird beim Zeichnen von Linien für jedes Null-Bit in der Linienmaske die Grafik-PAPER-INK benutzt, wenn der Hintergrund-Modus opaque (dekkend) ist.

BBE7 GRA GET PAPER Erfrage die momentane Hintergrund-Tinte.

Eingaben: keine
Ausgaben: A = Hintergrund-Tinte (Grafik-PAPER-INK)
Unverändert: BC, DE, HL, IX, IY

BBEA GRA PLOT ABSOLUTE Setze einen Punkt auf die angegebenen Position.

Eingaben: DE = X-Koordinate
HL = Y-Koordinate
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zum Origin und vorzeichenbehaftet (–32768 bis +32767).

Der Punkt wird mit der aktuellen Vordergrund-Tinte (Grafik-PEN-INK) gesetzt und dabei entsprechend dem Grafik-Vordergrund-Modus mit der Tinte (INK) des alten Punktes verknüpft.

Liegt der Punkt außerhalb des Grafikfensters, wird er nicht gesetzt.

Der Punkt wird gesetzt, indem IND GRA PLOT aufgerufen wird.

BBED GRA PLOT RELATIVE Setze einen Punkt relativ zur momentanen Position des Grafik-Cursors.

Eingaben: DE = X-Versatz
HL = Y-Versatz
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zur aktuellen Position des Grafik-Cursors und vorzeichenbehaftet (–32768 bis +32767).

Ansonsten wie &BBEA GRA PLOT ABSOLUTE.

BBF0 GRA TEST ABSOLUTE Teste, welche Tintennummer der Punkt auf der angegebenen Position hat.

Eingaben: DE = X-Koordinate des zu testenden Punktes
HL = Y-Koordinate des zu testenden Punktes
Ausgaben: A = Tinte (INK) des Punktes
Unverändert: IX, IY

Die Koordinaten sind relativ zum Origin und vorzeichenbehaftet (–32768 bis +32767).

Wird ein Punkt außerhalb des Grafikfensters getestet, so wird die aktuelle Hintergrund-Tinte (Grafik-PAPER-INK) zurückgegeben.

BBF3 GRA TEST RELATIVE Teste, welche Tintennummer ein Punkt relativ zur aktuellen Lage des Grafik-Cursors hat.

Eingaben: DE = X-Versatz
HL = Y-Versatz

Ausgaben: A = Tinte (INK) des Punktes
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zur aktuellen Position des Grafik-Cursors und vorzeichenbehaftet (–32768 bis 32767).

Wird ein Punkt außerhalb des Grafikfensters getestet, so wird die aktuelle Hintergrund-Tinte (Grafik-PAPER-INK) zurückgegeben.

BBF6 GRA LINE ABSOLUTE Ziehe eine Linie von der momentanen Position des Grafik-Cursors zur angegebenen absoluten Position.

Eingaben: DE = X-Koordinate des Zielpunktes
HL = Y-Koordinate des Zielpunktes
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zum Origin und vorzeichenbehaftet (–32768 bis +32767).

Von der aktuellen Position des Grafik-Cursors wird eine Linie zur angegebenen Position gezogen. Dafür wird die Vordergrund-Tinte (Grafik-PEN-INK) benutzt und bei allen Punkten entsprechend dem Grafik-Vordergrund-Modus mit der alten Tinte (INK) des Punktes verknüpft. Die Linie wird durch Aufruf von IND GRA LINE gezeichnet.

Beim CPC 664 und 6128 bestimmt noch zusätzlich die Linienmaske, wie die einzelnen Punkte der Linie gesetzt werden. Für jedes Null-Bit wird die Grafik-PAPER-Tinte und der Grafik-Hintergrund-Modus benutzt.

BBF9 GRA LINE RELATIVE Ziehe eine Linie zu einer Position relativ zur aktuellen Lage des Grafik-Cursors.

Eingaben: DE = X-Versatz
HL = Y-Versatz
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zur aktuellen Position des Grafik-Cursors und vorzeichenbehaftet (–32768 bis +32767). Ansonsten wie &BBF6 GRA LINE ABSOLUTE.

BBFC GRA WR CHAR Zeichne einen Buchstaben an der Position des Grafik-Cursors.

Eingaben: A = Zeichencode
Ausgaben: keine
Unverändert: IX, IY

Alle Zeichen werden ausgedruckt, Controlcodes werden nicht befolgt. Die Zeichen werden so ausgegeben, daß der Grafik-Cursor auf der linken, oberen Ecke der Zeichenmatrix liegt. Danach wird der Grafik-Cursor um einen Buchstaben nach rechts bewegt, also je nach Bildschirmmodus um 8, 16 oder 32 Koordinateneinheiten.

Zum Zeichnen des Zeichens wird IND SCR WRITE benutzt. Deshalb werden alle Einstellungen der Grafik-VDU befolgt, aber keine der Text-VDU (außer der TAG-Einstellung).

Diese Einstellungen sind:

- Grenzen des Grafikfensters
- Grafik-PEN-INK
- Grafik-PAPER-INK
- Grafikmodus (Verknüpfung mit der alten Tinte der Punkte)

Beim CPC 464 gibt es nur einen Grafikmodus für die Vordergrund-Pixel (der Hintergrund ist immer deckend/opaque), bei dem CPC 664 und 6128 muß man noch zwischen Vordergrund- und Hintergrund-Modus unterscheiden. Letzterer wird als Default wie beim CPC 464 auf opaque eingestellt. Die zweite Wahlmöglichkeit ist transparent. Hier werden die Hintergrund-Pixel einfach nicht geplottet.

BD43 GRA DEFAULT Stelle die Standardwerte für die verschiedenen Optionen der Grafik-VDU ein (nur CPC 664 und 6128).

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Diese Routine nimmt folgende Einstellungen vor:

Grafik-Vordergrund-Modus – opaque
Grafik-Hintergrund-Modus – opaque

Erster-Punkt-Option	– Ein
Linienmaske	– &FF (durchgezogene Linien)

Bis auf die erste Option, die auch beim CPC 464 wählbar ist, werden alle zusätzliche Optionen des CPC 664 und 6128 so eingestellt, daß die Grafikausgabe der des CPC 464 entspricht.

BD46 GRA SET BACK Setze den Hintergrund-Modus der Grafik-VDU (nur CPC 664 und 6128).

Eingaben:	A = Hintergrund-Modus
Ausgaben:	keine
Unverändert:	AF, BC, DE, HL, IX, IY

Mit diesem Vektor kann festgelegt werden, ob Hintergrund-Punkte gezeichnet werden sollen (opaque) oder nicht (transparent). Das entspricht dem Hintergrund-Modus der Text-VDU, bezieht sich hier jedoch auf die Hintergrund-Pixel in Linien (hervorgerufen durch eine entsprechende Einstellung des Maskenbytes) und in Buchstaben, wenn man &BBFC GRA WR CHAR (nicht beim CPC 464) benutzt.

A=0 → opaque (default)
A>0 → transparent

BD49 GRA SET FIRST Setze die Erster-Punkt-Option für die zu zeichnenden Linien (nur CPC 664 und 6128).

Eingaben:	A = Erster-Punkt-Flag
Ausgaben:	keine
Unverändert:	AF, BC, DE, HL, IX, IY

Speziell wenn der Grafik-Vordergrund-Modus auf XOR eingestellt ist, ist es sinnvoll, auch die Erster-Punkt-Option zu löschen (ersten Punkt einer Linie nicht zeichnen). Dadurch wird vermieden, daß in Streckenzügen die Eckpunkte zweimal gezeichnet und damit wieder gelöscht werden. Das wäre speziell dann äußerst ungünstig, wenn man einen geschlossenen Streckenzug nachher mit &BD52 GRA FILL ausmalen möchte. Der Algorithmus würde dann nämlich durch die Löcher an den Knickpunkten "auslaufen".

A=0 → ersten Punkt nicht zeichnen (aus)
A>0 → ersten Punkt zeichnen (ein) (default)

BD4C GRA SET LINE MASK Lege die Punktmaske für Linien neu fest (nur CPC 664 und 6128).

Eingaben: A = Punktmaske
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Mit Hilfe dieser Maske ist es möglich, entweder gebrochene (gepunktete) Linien oder auch zweifarbig gemusterte Linien zu zeichnen (nicht beim CPC 464).

Die Punktmaske ist Bit-signifikant. Wenn eine Linie gezeichnet wird, wird die Maske einmal rotiert und so das nächste Bit in ihr "angewählt". Ist diese gesetzt, wird der nächste Punkt der Linie im Grafik-Vordergrund-Modus gesetzt. Ist das Bit aber gleich 0, wird der Punkt im Hintergrund-Modus gesetzt.

Durch das Rotieren ist die nur 8 Bit breite Maske nicht bereits nach 8 Punkten verbraucht, sondern wird wieder von vorne abgearbeitet. Zwischen zwei aufeinanderfolgend gezeichneten Linien wird die Maske nicht wieder in Null-Position rotiert, wodurch es keinen Bruch im Muster gibt. Deswegen wird die Maske auch nicht für einen nicht gesetzten ersten Punkt einer Linie einmal im Leerlauf gedreht.

Leider optimieren die Linien-Algorithmen der Grafik-VDU ein wenig, wodurch das Ergebnis hinterher nicht mehr ganz so optimal ist: Alle eher vertikalen Linien ($|dX| < |dY|$) werden von unten nach oben und alle eher horizontalen Linien ($|dX| > |dY|$) werden de facto von links nach rechts gezeichnet und die Maske auch in dieser Richtung angewendet.

BD4F GRA FROM USER Konvertiere die User-Koordinaten (relativ zum Origin) in Basiskoordinaten (relativ zur linken unteren Ecke) (nur CPC 664 und 6128).

Eingaben: DE = User-X-Koordinate
HL = User-Y-Koordinate
Ausgaben: DE = Basis-X-Koordinate
HL = Basis-Y-Koordinate
Unverändert: BC, IX, IY

Die Basiskoordinaten sind die "Low Level-Koordinaten", die vor allem vom Screen Pack benutzt werden. Hierbei ist die linke untereEcke der Nullpunkt.

Die Skalierung richtet sich nach den physikalisch tatsächlich darstellbaren Punkten: In Y-Richtung geht sie von 0 bis 199, und in X-Richtung hängt sie vom Bildschirmmodus ab:

Modus 0: 0 bis 159
Modus 1: 0 bis 319
Modus 2: 0 bis 639

Leider wird im Schneider CPC hierfür eine äußerst interessante Art der Rundung benutzt: Es wird immer zum Origin hin gerundet! So liegen in Modus 1 beispielsweise die Punkte (–1,y), (0,y) und (1,y) alle auf demselben Pixel (mit konstantem x gilt dasselbe für die Y-Koordinate). Normalerweise kommen nur zwei Punkte auf jedes Pixel im Bildschirm.

Mit Ausnahme von X in Modus 2 (80 Zeichen) gibt es deshalb beim Null-Durchgang in X- oder Y-Richtung immer einen Stetigkeitssprung! Werden Kurvenzüge hier im XOR-Modus geplottet, so werden die Pixel auf dem Null-Durchgang oft doppelt geplottet und so wieder entfernt (ungünstig, wenn man nachher vielleicht noch mit FILL ausmalen will).

BD52 GRA FILL Male eine beliebige Fläche aus (nur CPC 664 und 6128).

Eingaben: A = Ausmalfarbe (nicht expandiert)
HL = Pufferadresse
DE = Pufferlänge
Ausgaben: CY=1 → Fläche vollständig ausgemalt
CY=0 → Fläche nicht oder nicht vollständig ausgemalt
Unverändert: IX, IY

Der Füllalgorithmus malt jede beliebige umrandete Fläche aus, ist sehr schnell und funktioniert sogar korrekt. Als Startpunkt gilt dabei die aktuelle Position des Grafik-Cursors. Dieser wird beim Ausmalen nicht verschoben. Als Füllgrenze gelten:

- das Grafikfenster
- Punkte, die in der Ausmalfarbe gesetzt sind
- Punkte, die in der Grafik-Vordergrund-Farbe gesetzt sind

Dabei gelten diagonal verbundene Punkte auch als Grenze, d.h. die von der Grafik-VDU gemalten Linien sind geeignet, die zu füllende Fläche zu begrenzen.

Ein CY=0 als Rückgabebedingung kann folgende Gründe haben: Entweder liegt der Grafik-Cursor außerhalb des Grafikfensters oder auf einem Pixel mit einer begrenzenden Tinte. Dann wird überhaupt nichts ausgemalt. Oder der zur Verfügung gestellte Puffer war zu klein, dann wurde die Fläche nicht vollständig ausgemalt.

Der Puffer dient dazu, Verzweigungspunkte der auszumalenden Fläche zu speichern. Für die meisten einfachen Formen kommt man dabei mit 10 Punkten aus. Bei komplizierteren Figuren muß man entsprechend mehr Platz zur Verfügung stellen. Die Fill-Routine benötigt dabei 7 Bytes pro Verzweigung (plus ein Byte für die Endkennung).

THE SCREEN PACK (SCR)
Die Bildschirm-Routinen

BBFF SCR INITIALISE Initialisiere das Screen Pack.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Komplette Initialisierung des Screen-Packs.

Betroffen sind: die Indirections des Screen Packs
alle Tinten (INKs) werden auf ihre Standardwerte gesetzt
die Blinkperioden ebenfalls
der Bildschirmmodus wird auf MODE 1 eingestellt
der Bildschirm wird ins obere RAM-Viertel gelegt
der Scroll-Offset wird auf 0 gesetzt
der Bildschirm wird mit Tinte 0 gelöscht
der Grafikmodus wird auf opaque (deckend) eingestellt
das Event zum Farbenblinken wird initialisiert

BC02 SCR RESET Setze das Screen Pack zurück.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Kleine Initialisierung des Screen Packs.

Betroffen sind: die Indirections des Screen-Packs
die INKs (Tinten) werden auf ihre Standardwerte gesetzt
die Blinkperioden werden auf ihre Standardwerte gesetzt
der Grafikmodus wird auf opaque (deckend) eingestellt.

BC05 SCR SET OFFSET Verändere den Hardware-Scroll-Offset des Bildschirms.

Eingaben: HL = neuer Scroll-Offset
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Der Scroll-Offset wird zuerst mit &07FE maskiert, um einen erlaubten Wert sicherzustellen. Dann wird der Bildschirm (Hardware-mäßig) gescrollt.

BC08 SCR SET BASE Verlege den Bildschirm in ein anderes RAM-Viertel.

Eingaben: A = MSB des RAM-Viertels
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Die Bildschirm-Basis wird zuerst mit &C0 maskiert, da nur exakte RAM-Viertel in Frage kommen. Dann wird die Hardware von der neuen Lage unterrichtet.

Der Scroll-Offset wird nicht geändert, der neue Bildschirm wird auch nicht gelöscht.

BC0B SCR GET LOCATION Erfrage Scroll-Offset und Speicherviertel des Bildschirms.

Eingaben: keine
Ausgaben: A = MSB des RAM-Viertels des Bildschirmspeichers
HL = Scroll-Offset
Unverändert: BC, DE, IX, IY

BC0E SCR SET MODE Lösche den Bildschirm, setze den Scroll-Offset auf Null-, und lege den Bildschirmmodus neu fest.

Eingaben: A = neuer Bildschirm-Modus
Ausgaben: keine
Unverändert: IX, IY

Zuerst wird A mit 3 maskiert. Enthält A danach den Wert 3, kehrt die Routine sofort zurück.

Sonst wird der Bildschirm mit INK 0 gelöscht und auf den neuen Modus (0, 1 oder 2) umgestellt.

Alle Fenster, auch das Grafikfenster, werden auf den ganzen Bildschirm ausgeweitet, die Cursor ausgeschaltet (OFF) und in die linke obere Ecke gestellt. Der Grafik-Cursor und Origin kommen in die linke untere Ecke.

Alle PEN- und PAPER-INKs, auch die der Grafik-VDU, werden entsprechend dem neuen Bildschirmmodus maskiert und können sich deshalb eventuell ändern.

BC11 SCR GET MODE Erfrage den momentan eingestellten Bildschirmmodus.

Eingaben: keine
Ausgaben: A = MODE und CY- und Z-Flag werden gesetzt
Unverändert: BC, DE, HL, IX, IY

Folgende Ausgaben sind möglich:

MODE = 0 → CY=1 und Z=0 und A=0
MODE = 1 → CY=0 und Z=1 und A=1
MODE = 2 → CY=0 und Z=0 und A=2

BC14 SCR CLEAR Lösche den ganzen Bildschirm mit der Tinte 0.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Der Bildschirm wird mit INK 0 gelöscht. Hat ein Fenster eine andere PAPER-INK, so wird das sichtbar, sobald dort ein Zeichen geschrieben oder das Fenster gelöscht wird.

Der Scroll-Offset wird auf 0 gesetzt.

Nachdem der Bildschirm gelöscht ist, wird das INK-FLASH-Event neu initialisiert.

BC17 SCR CHAR LIMITS Erfrage, wie viele Buchstaben, abhängig vom aktuellen Bildschirmmodus, in eine Zeile passen.

Eingaben: keine
Ausgaben: B = letzte Spalte des Bildschirms
C = letzte Zeile des Bildschirms
Unverändert: DE, HL, IX, IY

Die letzte Zeile und die letzte Spalte werden relativ zur linken oberen Ecke mit den Koordinaten (0,0) angegeben (physikalische Position). Die letzte Zeile ist immer 24, die letzte Spalte, abhängig vom Bildschirmmodus, 19, 39 oder 79.

BC1A SCR CHAR POSITION Konvertiere eine "physikalische" Buchstabenposition in die zugehörige Bildspeicheradresse.

Eingaben: H = Spalte
L = Zeile
Ausgaben: HL = Byte-Adresse
B = Buchstabenbreite in Bytes
Unverändert: C, DE, IX, IY

Spalte und Zeile werden relativ zur linken oberen Ecke mit der Koordinate (0,0) angegeben.

H und L werden nicht auf Plausibilität geprüft. Unsinnige Koordinaten erzeugen deshalb unsinnige Adressen. Die angegebene Adresse gilt für die linke obere Ecke des Buchstabens.

BC1D SCR DOT POSITION Konvertiere eine Grafikposition in "Basis"- Koordinaten in die zugehörige Bildspeicheradresse.

Eingaben: DE = X-Koordinate
HL = Y-Koordinate

Ausgaben: HL = Byte-Adresse
C = Bitmaske für den Punkt
B = Anzahl Punkte pro Byte – 1
Unverändert: IX, IY

Die X- und Y-Koordinate beziehen sich dabei auf die linke untere Ecke mit der Koordinate (0,0). Die Skalierung in X- und Y-Richtung ist abhängig vom Bildschirmmodus genau so, daß jedem realen Pixel im Bildschirm genau eine eigene Koordinate zufällt. Die rechte obere Ecke hat daher folgende Koordinaten:

Modus 0: (159,199)
Modus 1: (319,199)
Modus 2: (639,199)

Die übergebenen Koordinaten werden nicht auf Plausibilität geprüft. Unsinnige Koordinaten ergeben unsinnige Adressen.

BC20 SCR NEXT BYTE Bestimme die Adresse des Bytes im Bildschirmspeicher rechts von der angegebenen Adresse.

Eingaben: HL = Adresse im Bildschirmspeicher
Ausgaben: HL = Adresse des Bytes rechts daneben
Unverändert: BC, DE, IX, IY

Normalerweise muß dazu die Adresse nur inkrementiert werden. Durch die Möglichkeit, den Bildschirm aber auch Hardware-mäßig zu scrollen, muß auf das Ende der "RAM-Zeile" getestet werden.

Ist die übergebene Position die letzte in der Zeile, so ist die neue Adresse die erste Position im Bildschirm 8 Zeilen tiefer. Liegt diese nicht mehr im Bildschirm, so ist es das erste Byte in den unbenutzten 48 Bytes pro "RAM-Zeile".

BC23 SCR PREV BYTE Bestimme die Adresse des Bytes im Bildschirmspeicher links von der angegebenen Adresse.

Eingaben: HL = Adresse im Bildschirmspeicher
Ausgaben: HL = Adresse des Bytes links daneben
Unverändert: BC, DE, IX, IY

Normalerweise muß dazu die Adresse nur dekrementiert werden. Durch die Möglichkeit, den Bildschirm aber auch Hardware-mäßig zu scrollen, muß auf den Anfang einer RAM-Zeile getestet werden.

Ist die übergebene Position die erste in einer Zeile, so ist die neue Adresse die letzte Position im Bildschirm 8 Zeilen höher. Liegt diese nicht mehr im Bildschirm, so ist es das letzte Byte in den unbenutzten 48 Bytes pro "RAM-Zeile".

BC26 SCR NEXT LINE Bestimme die Adresse des Bytes im Bildschirmspeicher unter der angegebenen Adresse.

Eingaben: HL = Adresse im Bildschirmspeicher
Ausgaben: HL = Adresse darunter
Unverändert: BC, DE, IX, IY

Wird eine Adresse in der letzten Bildschirmzeile übergeben, so ergibt sich kein sinnvolles Ergebnis.

BC29 SCR PREV LINE Bestimme die Adresse des Bytes im Bildschirmspeicher über der angegebenen Adresse.

Eingaben: HL = Adresse im Bildschirmspeicher
Ausgaben: HL = Adresse darüber
Unverändert: BC, DE, IX, IY

Wird eine Adresse in der ersten Zeile des Bildschirms übergeben, so ergibt sich kein sinnvolles Ergebnis.

BC2C SCR INK ENCODE Konvertiere eine Tintennummer in ein Byte, das, in den Bildschirm gepoket, alle betroffenen Punkte in dieser Tinte darstellt.

Eingaben: A = Tintennummer (INK)
Ausgaben: A = Farb-Byte
Unverändert: BC, DE, HL, IX, IY

Das so erhaltene Byte kann mit einer Bitmaske für das gewünschte Pixel maskiert werden, um so ein Byte zu erhalten, das nur noch das eine Pixel in der gewünschten Tinte eingefärbt enthält. Die Farb-Bytes, die einer Tinte zugeordnet sind, sind in jedem Bildschirmmodus anders.

BC2F SCR INK DECODE Bestimme die Tintennummer des ersten Punktes von links im übergebenen Byte.

Eingaben: A = Farb-Byte
Ausgaben: A = Tintennummer (INK)
Unverändert: BC, DE, HL, IX, IY

Die Farbcodierung des ersten (linken) Pixels im übergebenen Byte wird entsprechend dem Bildschirmmodus in die Tintennummer zurückkonvertiert.

BC32 SCR SET INK Lege die beiden Farben fest, in der eine Tinte in den beiden Blinkperioden dargestellt werden soll.

Eingaben: A = Tintennummer (INK)
B und C sind die beiden zugehörigen Farben
Ausgaben: keine
Unverändert: IX, IY

Die Tintennummer wird mit &F und die Farben werden mit &1F maskiert, um sie in erlaubte Grenzen zu zwingen. Ab dem nächsten Strahlrücklauf werden alle Punkte dieser Tinte in den neuen Farben angezeigt. B ist die erste und C die zweite Farbe. Werden zwei gleiche Farben angegeben, so blinkt diese Tinte nicht.

BC35 SCR GET INK Erfrage die Farben, in der eine Tinte in den beiden Blinkperioden dargestellt wird.

Eingaben: A = Tintennummer (INK)
Ausgaben: B und C enthalten die beiden Farben
Unverändert: IX, IY

Die Tintennummer wird mit &F maskiert, um einen gültigen Wert sicherzustellen. B ist die erste und C die zweite Farbe.

BC38 SCR SET BORDER Bestimme die Farben, in denen der Bildschirmrand in den beiden Blinkperioden dargestellt wird.

Eingaben: B und C enthalten die beiden Farben
Ausgaben: keine
Unverändert: IX, IY

Die Farben werden mit &1F maskiert, um sie in erlaubte Grenzen zu zwingen. Ab dem nächsten Strahlrücklauf wird der Bildschirmrand (BORDER) in den neuen Farben angezeigt. B ist die erste und C die zweite Farbe. Werden zwei gleiche Farben angegeben, so blinkt der BORDER nicht.

BC3B SCR GET BORDER Erfrage die Farben, in denen der Bildschirmrand in den beiden Blinkperioden dargestellt wird.

Eingaben: keine
Ausgaben: B und C enthalten die beiden Farben.
Unverändert: IX, IY

BC3E SCR SET FLASHING Lege die Länge der beiden Blinkperioden neu fest.

Eingaben: H und L enthalten die Längen der beiden Perioden
Ausgaben: keine
Unverändert: BC, DE, IX, IY

H enthält die anzuzeigende Dauer für die erste Farbe, L für die zweite. Die Zeiten werden in Monitorbildern angegeben, also je nach Fernsehnorm in 1/50stel oder 1/60stel Sekunden. Ein Wert von 0 wird als 256 interpretiert.

Die neuen Zeiten werden erst nach dem nächsten Blinkwechsel übernommen.

BC41 SCR GET FLASHING Erfrage, welche Länge die beiden Blinkperioden momentan haben.

Eingaben: keine
Ausgaben: H und L enthalten die Länge der beiden Perioden
Unverändert: BC, DE, IX, IY

Siehe &BC3E SCR SET FLASHING.

BC44 SCR FILL BOX Fülle einen rechteckigen Bildschirmausschnitt mit einer Tinte. Die Grenzen werden in Buchstabenpositionen angegeben.

Eingaben: A = encoded INK (Farb-Byte)

Eingaben: H = linke Spalte
D = rechte Spalte
L = oberste Zeile
E = unterste Zeile
Ausgaben: keine
Unverändert: IX, IY

Die anzugebenden Grenzen sind die inneren Grenzen der Füllfläche und beziehen sich auf die linke obere Ecke mit den Koordinaten (0,0). Die Grenzen werden nicht auf Plausibilität kontrolliert. Unmögliche Grenzwerte führen deshalb zu unmöglichen Ergebnissen.

BC47 SCR FLOOD BOX Fülle einen rechteckigen Bildschirmausschnitt mit einer Tinte. Die Grenzen werden in Byte-Positionen angegeben.

Eingaben: C = Encoded INK (Farb-Byte)
HL = Adresse des Bytes in der linken oberen Ecke
D = Breite der Füllfläche in Bytes
E = Höhe der Füllfläche in Pixel-Zeilen
Ausgaben: keine
Unverändert: IX, IY

Die übergebenen Werte werden nicht auf Plausibilität überprüft. Unmögliche Grenzen führen zu unsinnigen Ergebnissen. Die Angaben in den Registern D und E verstehen sich ohne Vorzeichen, eine 0 bedeutet 256.

BC4A SCR CHAR INVERT Invertiere eine Buchstabenposition (→ Cursor).

Eingaben: B und C enthalten je eine encoded INK (Farb-Byte)
H = Spalte
L = Zeile
Ausgaben: keine
Unverändert: IX, IY

Die Spalte und Zeile werden relativ zur linken oberen Ecke mit den Koordinaten (0,0) angegeben. Die Koordinaten werden nicht auf Plausibilität überprüft. Zu große Werte erzeugen unsinnige Ergebnisse.

Alle Bytes der angegebenen Buchstabenposition werden wie folgt verändert:

neu = alt xor B xor C

Mit diesem Effekt werden die Standard-Cursor-Flecken des CPC erzeugt. Durch nochmaliges Aufrufen dieser Routine wird der Flecken automatisch wieder entfernt (Cursor-Setzen und- Entfernen wird mit derselben Routine erledigt!).

BC4D SCR HW ROLL Scrolle den ganzen Bildschirm Hardware-mäßig um eine Buchstabenposition rauf oder runter.

Eingaben: B=0 → Bildschirm nach unten scrollen
B>0 → Bildschirm nach oben scrollen
A = encoded INK (Farb-Byte)
Ausgaben: keine
Unverändert: IX, IY

Der Bildschirm wird Hardware-mäßig um 8 Pixel-Zeilen gescrollt. Die neu sichtbar werdende Buchstabenzeile wird mit dem angegebenen Farb-Byte gelöscht.

Zum Hardware-mäßigen Scrollen wird der Scroll-Offset um 80 erhöht oder vermindert (MOD &800). Außerdem findet die ganze Aktion erst zum nächsten Strahlrücklauf statt. Bis dahin wartet die Routine.

BC50 SCR SW ROLL Scrolle einen Bildschirmausschnitt um eine Buchstabenposition rauf oder runter.

Eingaben: B=0 → Scrollen des Bildschirmausschnitts nach unten
B>0 → Scrollen nach oben
A = encoded INK (Farb-Byte)
H = linke Spalte
D = rechte Spalte
L = oberste Zeile
E = unterste Zeile
Ausgaben: keine
Unverändert: IX, IY

Die Grenzen des zu scrollenden Bildschirmausschnitts sind innere Grenzen und beziehen sich auf die linke obere Ecke mit den Koordinaten (0,0). Die Grenzen werden nicht auf Plausibilität kontrolliert. Unvorhersehbare Effekte können auftreten, wenn man unsinnige Werte übergibt.

Der Ausschnitt wird um 8 Pixel-Zeilen gescrollt. Die freiwerdende Zeile wird mit dem angegebenen Farb-Byte gelöscht.

BC53 SCR UNPACK Expandiere eine Zeichenmatrix entsprechend dem momentanen Bildschirmmodus.

Eingaben: HL zeigt auf die Zeichenmatrix und
DE zeigt auf einen Puffer für die expandierten Bytes (8, 16 oder 32 Bytes)
Ausgaben: Bytes im Puffer
Unverändert: IX, IY

Die Zeichenmatrix wird je nach Bildschirmmodus in 8, 16 oder 32 Bytes expandiert, die, in die Zeichenposition auf dem Bildschirm gepoket, dort das Zeichen mit INK 15, 3 oder 1 (je nach Bildschirmmodus) auf INK 0 erscheinen ließe. Die expandierten Bytes können jedoch vorher noch mit encoded INKs (Farb-Bytes) und, je nach Hintergrund-Modus, mit dem alten Bildschirminhalt verknüpft werden.

BC56 SCR REPACK Komprimiere die expandierten Bytes wieder zu einer Zeichenmatrix.

Eingaben: A = encoded INK (Farb-Byte)
H = Spalte des Zeichens
L = Zeile des Zeichens
DE zeigt auf den Puffer (8 Bytes)
Ausgaben: Matrix im Puffer
Unverändert: IX, IY

Die Spalte und Zeile werden relativ zur linken oberen Ecke des Bildschirms mit den Koordinaten (0,0) in Buchstabenpositionen angegeben. Sie werden nicht auf Plausibilität geprüft. Der Versuch, von außerhalb des Bildschirms ein Zeichen zu lesen, führt zu keinem sinnvollen Ergebnis.

In der erzeugten Zeichenmatrix sind all die Bits gesetzt, deren Pendant auf dem Bildschirm in der durch das Farb-Byte angegebenen Tinte gesetzt waren.

BC59 SCR ACCESS Setze den Zeichenmodus für die Grafik-VDU.

Eingaben: A = neuer Modus
Ausgaben: keine
Unverändert: IX, IY

Der Wert im Register A wird mit 3 maskiert, um einen gültigen Wert zu erzwingen.

Der Zeichenmodus (Grafikmodus) bestimmt, wie beim Zeichnen die neue Tinte für einen Punkt aus dessen alter und der neuen Tinte berechnet wird:

A=0 →	opaque, deckend (force):	Neu := INK
A=1 →	exclusiv-oder:	Neu := Alt xor INK
A=2 →	und:	Neu := Alt and INK
A=3 →	oder:	Neu := Alt or INK

Der Grafikmodus beeinflußt nur IND SCR WRITE. Diese Indirection wird nur von den PLOT- und DRAW-Routinen aufgerufen und bei der Ausgabe eines Zeichen auf der Position des Grafik-Cursors.

BC5C SCR PIXELS Setze einen Punkt im Bildschirm.

Eingaben:	B = encoded INK (Farb-Byte)
	C = Masken-Byte für das (oder die) Pixel
	HL = Adresse im Bildschirmspeicher
Ausgaben:	keine
Unverändert:	BC, DE, HL, IX, IY

Die Bildschirmadresse wird nicht überprüft. Unsinnige Werte führen zu unvorhersehbaren Ergebnissen.

Das oder die durch C ausmaskierten Pixel werden mit der Farbe in B an der Adresse HL in den Bildschirm gesetzt.

BC5F SCR HORIZONTAL Zeichne eine waagerechte Linie.

Eingaben:	A = encoded INK (Farb-Byte)
	DE = linke X-Koordinate
	BC = rechte X-Koordinate
	HL = Y-Koordinate
Ausgaben:	keine
Unverändert:	IX, IY

Die Koordinatenangaben erfolgen in Basiskoordinaten. Siehe dazu &BC1D SCR DOT POSITION.

Die übergebenen Koordinaten müssen im Bildschirm liegen, und außerdem muß DE kleiner oder gleich BC sein. Wird das nicht beachtet, können unvorhergesehene Effekt auftreten.

BC62 SCR VERTICAL Zeichne eine senkrechte Linie.

Eingaben: A = encoded INK (Farb-Byte)
DE = X-Koordinate des unteren Endes
HL = Y-Koordinate des unteren Endes
BC = Y-Koordinate des oberen Endes
Ausgaben: keine
Unverändert: IX, IY

Die Angaben erfolgen in Basiskoordinaten. Siehe dazu &BC1D SCR DOT POSITION.

Die übergebenen Koordinaten müssen im Bildschirm liegen. Außerdem muß HL kleiner oder gleich BC sein. Andernfalls können unvorhergesehene Effekte auftreten.

BD55 SCR SET POSITION Lege die Lage des Bildschirms nur für die Software neu fest (nur CPC 664 und 6128).

Eingaben: HL = Scroll-Offset
A = Bildschirm-Basis
Ausgaben: A und HL legalisiert
Unverändert: BC, DE, IX, IY

Dieser Vektor ändert Lage und Scroll-Zustand nur für die Software, d.h. für das Screen Pack, die Text und die Grafik-VDU. Die Bildausgabe erfolgt nach wie vor aus dem zuvor gewählten Speicherviertel mit dem alten Scroll-Zustand.

Dadurch ist es beispielsweise möglich, ein neues Bild unsichtbar aufzubauen, während das alte noch angezeigt wird. Beim CPC 664 kommt dabei praktischerweise allerdings nur die Bank 1 (&4000 bis &7FFF) in Frage, beim CPC 6128 kann man zusätzlich auch noch an der RAM-Konfiguration herumspielen.

A wird als oberes Adreß-Byte benutzt, um das Speicherviertel festzulegen, und wird deshalb mit &C0 maskiert. Die Maske für HL ist &07FE.

Besitzer des CPC 464 können ersatzweise folgende Systemvariablen verändern:

```
&B1C9,&B1CA - SCR OFFSET   (HL)
      &B1CB - SCR BASE     (A)
```

THE CASSETTE MANAGER (CAS)
Die Kassetten- (und Disketten-) Routinen

Die von AMSDOS gepatchten Routinen sind mit * markiert.

Diese Top-Level-Routinen können, bis auf CAS OUT ABANDON und CAS IN ABANDON, einen Fehler zurückmelden. Dabei gibt es jedoch zum Teil erhebliche Unterschiede zwischen CAS-464, CAS-664/6128 und AMSDOS.

Gemeinsam ist: Bei erfolgreicher Beendigung einer Opertion wird das CY-Flag gesetzt:

AMSDOS, CAS-464, CAS-664/6128: CY=1 → o.k.

Die Kassetten-Operationen können gebreakt werden. Das wird beim CPC 464 laut Firmware-Manual mit CY=0 und Z=1 vermerkt, bei CPC 664 und 6128 mit dem Fehlercode 0 im A-Register. Trotzdem scheinen beide real gleich zu arbeiten:

CAS-464, CAS-664/6128: CY=0 und Z=1 und A=0 → Break

Disketten-Operationen können nicht unterbrochen werden.

Alle anderen Fehler werden bei den Kassetten-Routinen immer mit CY=0 und Z=0 angemerkt. Der CPC 664/6128 gibt in A aber immer noch einen Fehler-Code aus.

Datei nicht eröffnet bzw. bereits eine Datei eröffnet:	CPC 464: CY=0, Z=0 A = 255 oder zwischen 1 und 5 CPC 664/6128: CY=0, Z=0 A = 14
End of File:	CPC 464: CY=0, Z=0 A zwischen 1 und 5 CPC 664/6128: CY=0, Z=0 A = 15

AMSDOS kennt noch mehr Fehler, die auch mit CY=0 gemeldet werden. Diese beiden Fehler werden wie bei den Kassetten-Routinen mit Z=0 gemeldet, die anderen aber mit Z=1, was beim Kassettenbetrieb der Break-Meldung entspricht

Die Folgende Tabelle enthält eine Aufstellung der möglichen Fehler, wie sie ganz allgemein beim CASSETTE MANAGER des CPC 464, beim CPC 664/6128 und unter AMSDOS auftreten können.

Zero-Flag	Fehlernummer im Akku	Bedeutung (A=AMSDOS, 4=CAS 464, 6=CAS664/6128)	
Z=1	0	4 6	Break
Z=0	1 ... 5, 255	4	End of File oder Datei nicht eröffnet oder bereits eine Datei eröffnet
Z=0	14	A 6	Datei nicht eröffnet bzw. bereits eine Datei eröffnet
Z=0	15	A 6	End of File (hard end)
Z=0	26	A	End of File (soft end)
Z=1	16	A	unbekannter Befehl, illegaler Dateiname, bad command
Z=1	17	A	Datei existiert bereits
Z=1	18	A	Datei existiert nicht
Z=1	19	A	Inhaltsverzeichnis ist voll
Z=1	20	A	Diskette ist voll
Z=1	21	A	Diskette wurde bei offener Datei gewechselt
Z=1	22	A	Datei ist schreibgeschützt
Z=1	Bit 6 gesetzt:	A	FDC-Fehler
Z=1	Bit 7 gesetzt:	A	Fehler wurde dem Anwender bereits mitgeteilt

FDC-Fehler: Bit 6 (&40) ist gesetzt. Die folgenden Bits bedeuten:

&01	1	– ID- oder Data-Adreßmarke fehlt	(Diskette ist nicht formatiert)
&02	2	– Diskette ist schreibgeschützt	(Schreibschutzkerbe ist geöffnet)
&04	4	– Sektor nicht auffindbar	(falsches Diskettenformat eingeloggt)
&08	8	– Laufwerk ist nicht bereit	(Diskette nicht/nicht richtig drin)
&10	16	– Pufferüberlauf	(dürfte nie vorkommen)
&20	32	– Kontrollsummen-Fehler	(Diskette möglicherweise beschädigt)

Die Fehler-Rückmeldung bei CAS CAT weicht stark von diesem allgemeinen Muster ab und ist bei dem Vektor selbst erläutert.

BC65 CAS INITIALISE Initialisiere die Kassetten-Routinen.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Komplette Initialisierung der Kassetten-Routinen.

Die Eingabe- und Ausgabedatei wird als geschlossen markiert. Die Schreibgeschwindigkeit wird auf 1000 Baud eingestellt, und die Meldungen der Kassetten-Routinen werden zugelassen.

*** nur CPC 664 und 6128: *** Der Kassettenmotor wird ausgeschaltet.

BC68 CAS SET SPEED Lege die Schreibgeschwindigkeit neu fest.

Eingaben: HL = Halbe Periodenlänge eines 0-Bits
A = Vorkompensation
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Die einzelnen Bits werden als jeweils eine Periode (0- und 1-Pegel) eines Rechtecksignals auf der Kassette aufgezeichnet. Die Periode für ein 1-Bit ist dabei doppelt so lang wie für eine 0. HL enthält die Länge einer halben 0-Periode (Länge des 0- oder 1-Pegels) in Mikrosekunden. Hier sind Werte zwischen 130 und 480 zulässig.

Da die analoge Aufzeichnungselektronik dazu neigt, die Signalflanke zwischen einem 0- und einem 1-Bit auf dem Band so zu verschieben, daß der Unterschied in der Periodenlänge verringert wird, wird das bereits beim Aufzeichnen durch eine Präkompensation ausgeglichen. A gibt dabei an, um wie viele Mikrosekunden die kurze Periode verkürzt und die lange noch verlängert wird.

Folgende Werte werden standardmäßig benutzt:

1000 Baud: HL=333 und A=25
2000 Baud: HL=167 und A=50

BC6B CAS NOISY Lege fest, ob die Kassetten-Routinen ihre Meldungen unterdrücken sollen.

Eingaben: A = Ein- oder Ausschalt-Flag
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Die Meldungen der Kassetten-Routinen werden unterdrückt, wenn in A ein Wert ungleich Null übergeben wird.

Betroffen sind: Press PLAY then any key:
Press REC and PLAY then any key:
Found <dateiname> block <nummer>
Loading <dateiname> block <nummer>
Saving <dateiname> block <nummer>

Diese Fehlermeldungen werden nicht beeinflußt: Read error <code>
Write error <code>
Rewind tape

BC6E CAS START MOTOR Starte den Motor des Kassettenrecorders.

Eingaben: keine
Ausgaben: A = alter Zustand des Motor-Relais
CY =1 → o.k., CY=0 → ESC
Unverändert: BC, DE, HL, IX, IY

Diese Routine startet den Motor immer. Lief er vorher noch nicht, wird ca. 2 Sekunden gewartet, außer wenn ESC gedrückt wird. Das signalisiert die Routine mit dem CY-Flag.

BC71 CAS STOP MOTOR Halte den Motor des Kassettenrecorders an.

Eingaben: keine
Ausgaben: A = alter Zustand des Motor-Relais
CY =1 → o.k. / CY=0 → ESC
Unverändert: BC, DE, HL, IX, IY

Diese Routine stoppt den Motor immer.

BC74 CAS RESTORE MOTOR Starte oder stoppe den Motor entsprechend einer früheren Einstellung.

Eingaben: A = alter Zustand des Motor-Relais
Ausgaben: CY =1 → o.k., CY=0 → ESC
Unverändert: BC, DE, HL, IX, IY

Der Motor wird immer entsprechend A gestoppt oder gestartet. Wie bei &BC6E CAS START MOTOR wird auf die Hochlaufzeit verzichtet, wenn ESC gedrückt wird. Das wird in beiden Fällen mit dem CY-Flag angezeigt.

BC77 * CAS IN OPEN Eröffne eine Datei zur Eingabe.

Eingaben: B = Länge des Dateinamens
HL = Adresse des Dateinamens
DE = Adresse des 2k-Input-Puffers

Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k.
A = Dateityp
BC = logische Dateilänge
DE = Einsprungsadresse (bei Maschinencode)
HL = Adresse des 64-Byte-Puffers mit dem File-Header
Unverändert: IY

Der Dateiname kann im gesamten RAM liegen. Kleinbuchstaben werden in Großbuchstaben umgewandelt.

Der Puffer kann ebenfalls im gesamten RAM liegen und wird so lange benutzt, bis die Datei mit CAS IN CLOSE oder CAS IN ABANDON wieder geschlossen wird.

Die ersten 2 kByte der Datei werden sofort gelesen, wodurch auch sofort der File-Header zur Verfügung steht. ASCII-Dateien auf Diskette haben keinen Header!

BC7A * CAS IN CLOSE Schließe eine Eingabedatei.

Eingaben: keine
Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k.
Unverändert: IX, IY

Nach Aufruf dieser Routine kann eine neue Eingabedatei eröffnet werden. Außerdem wird der Input-Puffer freigegeben.

BC7D * CAS IN ABANDON Vergiß, daß eine Datei zur Eingabe eröffnet ist.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Diese Routine wirkt wie &BC7A CAS IN CLOSE, ist aber hauptsächlich für den Fehlerfall gedacht.

BC80 • CAS IN CHAR Lies ein Zeichen aus der Eingabedatei.

Eingaben: keine
Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k. A = Zeichen.
Unverändert: BC, DE, HL, IY

Dateien können nur zeichenweise oder nur als Block gelesen werden. Ein Mischen der beiden Lesearten ist nicht möglich.

*** Nur AMSDOS: ***

Das Ende von ASCII-Dateien wird normalerweise mit dem Zeichen 26 gekennzeichnet (CTRL-Z – Soft End). Durch einen Fehler im AMSDOS signalisiert diese Routine auch bei Binärdateien ein EOF, wenn dieses Zeichen gelesen wird. Das ist in einem Maschinesprache-Programm leicht abfangbar. Ist CY=0, muß man nur testen, ob A diesen Wert enthält. Dann muß man die Fehlermeldung einfach ignorieren.

BC83 • CAS IN DIRECT Lies die Eingabedatei in einem Zug.

Eingaben: HL = Startadresse im RAM, ab der die Datei eingelesen werden soll
Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k. und HL = Einsprungadresse
Unverändert: IY

CAS IN DIRECT kann nur direkt nach &BC77 CAS IN OPEN aufgerufen werden. Es darf noch kein Zeichen mit &BC80 CAS IN CHAR gelesen worden sein.

*** Nur AMSDOS: ***

ASCII- oder CPM-Dateien von der Diskette können nicht als Block gelesen werden, da sie keinen Header haben.

BC86 • CAS RETURN Gib das zuletzt gelesene Zeichen noch einmal zurück.

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Mit diesem Vektor kann das zuletzt mit &BC80 CAS IN CHAR gelesene Zeichen zurückgegeben werden. Es kann nur ein Zeichen zurückgegeben werden, und es muß auch bereits ein Zeichen gelesen worden sein. Nach dem Test auf EOF kann das Zeichen "Soft End" (Fehlercode 26) zurückgegeben werden, "Hard End" oder "Datei nicht eröffnet" (Fehlercodes 14 und 15) aber nicht.

BC89 * CAS TEST EOF Frage an, ob das Ende der Eingabedatei erreicht ist.

Eingaben: keine
Ausgaben: CY = 0 → Fehler. (Beschreibung am Anfang dieses Packs.)
CY = 1 → alles o.k. (kein EOF)
Unverändert: BC, DE, HL, IY

Das File-Ende kann (sinnvollerweise) nur bei zeichenweiser Eingabe getestet werden. Dies geschieht, indem ein Zeichen gelesen und, sofern kein "hard-" oder "softend" erkannt wird, wieder zurückgegeben wird.

BC8C * CAS OUT OPEN Eröffne eine Datei zur Ausgabe.

Eingaben: B = Länge des Dateinamens
HL = Adresse des Dateinamens im RAM
DE = Adresse eines 2k-Output-Puffers
Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k. HL zeigt auf den zukünftigen File-Header
Unverändert: IY

Der Output-Puffer wird nur benutzt, wenn die Datei zeichenweise geschrieben wird. AMSDOS testet mit Aufruf dieser Routine sofort, ob im angesprochenen Laufwerk eine Diskette eingelegt ist.

Im Fileheader können dann unter anderem folgende Felder beschrieben werden:

- Dateityp (default: ASCII-Datei)
- logische Dateilänge
- Einsprungadresse

ASCII-Dateien auf Diskette werden ohne Header abgespeichert.

BC8F * CAS OUT CLOSE Schließe die Ausgabedatei.

Eingaben: keine
Ausgaben: CY = 0 → Fehler. (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k.
Unverändert: IY

Jede Ausgabedatei muß mit CAS OUT CLOSE geschlossen werden, auch wenn sie mit &BC98 CAS OUT DIRECT geschrieben wurde. Sonst wird der letzte Block nicht auf Band gespeichert, bzw. die Datei nicht ins Inhaltsverzeichnis der Diskette eingetragen.

Trat ein Fehler auf (ESC), so wird die Datei nicht korrekt geschlossen. Nachdem diese Routine aufgerufen wurde, kann der Output-Puffer wieder für andere Zwecke benutzt werden.

BC92 * CAS OUT ABANDON Vergiß die Ausgabedatei. Hierbei gehen immer größere Datenmengen verloren.

Eingaben: keine
Ausgaben: keine
Unverändert: *IX, IY*

Diese Routine ist für den Fehlerfall gedacht. Der letzte Datenblock wird nicht gespeichert und, bei AMSDOS, die Datei nicht ins Inhaltsverzeichnis der Diskette eingetragen. Wie bei &BC8F CAS OUT CLOSE kann nachher der Output-Puffer wieder benutzt und eine neue Ausgabedatei eröffnet werden.

BC95 * CAS OUT CHAR Schreibe ein Zeichen in die Ausgabedatei.

Eingaben: A = Zeichencode
Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k.
Unverändert: BC, DE, HL, IY

Diese Routine kann nicht gemischt mit &BC98 CAS OUT DIRECT benutzt werden.

BC98 * CAS OUT DIRECT Schreibe die Ausgabe-Datei in einem Zug.

Eingaben: A = Dateityp
BC = Einsprungsadresse

DE = Länge des abzuspeichernden Bereiches
HL = Startadresse im RAM.
Ausgaben: CY = 0 → Fehler (Beschreibung am Anfang dieses Packs.)
CY = 1 → o.k.
Unverändert: IY

CAS OUT DIRECT darf pro Datei nur einmal aufgerufen werden (man kann die Datei nicht in mehreren Abschnitten speichern). Außerdem kann dieser Vektor nicht mit &BC95 CAS OUT CHAR gemischt benutzt werden.

BC9B * CAS CATALOG Erstelle ein Inhaltsverzeichnis.

Eingaben: DE = Adresse eines 2k-Puffers
Ausgaben: CY, Z und A enthalten Informationen über Erfolg oder Fehler.
Unverändert: DE, IY

CAS CATALOG benutzt den Input-Strom, deshalb darf keine Eingabedatei eröffnet sein. BASIC schließt eine evtentuell geöffnete Eingabedatei. Nach Aufruf dieses Vektors ist der Input-Strom geschlossen (es kann sofort eine Datei zum Lesen eröffnet werden), und die Kassettenmeldungen sind zugelassen.

Die Ausgabe-Bedingungen für die Katalog-Routinen unterscheiden sich unverständlicherweise stark von denen der anderen High-Level-Routinen.

Erfolgreiche Kataloge:

Der Katalog von Kassette muß gebreakt werden. Trotzdem wird das nicht als Fehler mit CY=0 angemerkt. Im Gegensatz dazu muß ein Katalog von Diskette nicht gebreakt werden. Dafür merkt aber jetzt AMSDOS einen Fehler an! Möglicherweise wollte man damit zum Kassetten-Katalog kompatibel sein und nahm für dessen Return-Bedingung (fälschlicherweise) das Standard-Verhalten Break=Fehler<->CY=0 an.

CAS CAT o.k.: CY=1
DISC CAT o.k.: CY=0 und Z=0

Fehler: Input-Stream in use:

CAS CAT 464: CY=0, Z=0, A=1 (1...5)
CAS CAT 664/6128: CY=0, Z=0, A=14 = Fehlercode
DISC CAT: o.k., der Strom wird vorher geschlossen.

Fehler: Hardware-Fehler:

CAS CAT: Lesefehler werden im Katalog vermerkt. Führt nicht zum Abbruch.
DISC CAT: CY = 0, Z = 1, A = Fehlercode

BC9E CAS WRITE Schreibe einen Speicherbereich auf die Kassette.

Eingaben: HL = Startadresse des Datenblocks im RAM
DE = Länge des Blocks
A = Synchronisationszeichen
Ausgaben: CY = 1 → o.k.
CY = 0 → Fehler, A enthält einen Fehlercode
Unverändert: IY

Diese und die folgenden beiden Routinen sind die "Low-Level-Routinen" des Cassette Managers. Mit dieser Routine können beliebig lange Blocks zusammenhängend auf Band gespeichert werden. Die Blocklänge 0 wird dabei als 65536 interpretiert.

Die möglichen Fehlercodes sind:

0 – Break
1 – Write error a: Overrun (zu kleine Werte in &BC68 CAS SET SPEED angegeben)

Das Synchronisationszeichen wird von den normalen Datei-Routinen benutzt, um hier mit verschiedenen Werten den Header und den Datenblock unterscheiden zu können:

Sync = &2C – Header
Sync = &16 – Daten

Alle drei Routinen frieren für die Dauer ihrer Arbeit die Tonausgabe ein, verbieten in dieser Zeit Interrupts, schalten das Motor-Relais automatisch ein und restaurieren dessen alten Zustand nachher wieder. Außerdem arbeiten sie generell nur im RAM.

BCA1 CAS READ Lies einen Speicherbereich von Kassette ein.

Eingaben: HL = Zieladresse für die Daten im RAM
DE = maximale Anzahl Bytes, die gelesen werden sollen
A = Synchronisationszeichen

Ausgaben: CY = 1 → o.k.
CY = 0 → Fehler oder ESC, A enthält einen Fehlercode
Unverändert: IY

Es ist nicht unbedingt nötig, einen kompletten Block von Kassette zu lesen. Mit DE kann man die Maximallänge festlegen. Ist der Block auf dem Band allerdings kürzer, wird im allgemeinen Fehler 1 erzeugt. Folgende Fehlercodes sind möglich:

0 – Break
1 – Read error a: gelesene Periode auf Kassette war unleserlich lang
2 – Read error b: Prüfsummenfehler

Ansonsten gilt dasselbe wie bei &BC9E CAS WRITE.

BCA4 CAS CHECK Vergleiche die Daten auf der Kassette mit einem Speicherbereich.

Eingaben: HL = Startadresse der Vergleichsdaten im RAM
DE = Länge des Blocks
A = Synchronisationszeichen
Ausgaben: CY = 1 → o.k.
CY = 0 → Fehler, A = Fehlercode
Unverändert: IY

Die möglichen Fehlercodes sind dieselben wie bei &BCA1 CAS READ, zusätzlich noch der folgende:

3 – Read error c: Im RAM steht was anderes, als gelesen wurde.

Ansonsten gilt wieder das bei &BC98 CAS WRITE gesagte.

THE SOUND MANAGER (SOUND)
Die Lautsprecher-Routinen

BCA7 SOUND RESET Setze die Lautsprecher-Routinen zurück.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Diese Routine stoppt die Tonausgabe, leert die Ton-Warteschlangen und setzt ein ON SQ GOSUB zurück. Die Hüllkurven bleiben erhalten.

BCAA SOUND QUEUE Schicke einen Ton zum Sound-Manager.

Eingaben: HL zeigt auf Parameterblock
Ausgaben: CY = 1 → Ton wurde in die Warteschlange aufgenommen, sonst CY = 0
Unverändert: IY, Wenn CY = 0, dann auch HL

Der Parameterblock:

DEFB Kanalstatus
DEFB Nr. der ENV (Volumen-Hüllkurve. 0 = keine Änderung)
DEFB Nr. der ENT (Frequenz-Hüllkurve. 0 = keine Änderung)
DEFW Periodenlänge (Frequenz)
DEFB Rauschperiode (0 = kein Rauschen)
DEFB Startamplitude
DEFW Dauer (in 1/100 sec.) oder Wiederholfaktor

Die Angaben entsprechen denen im SOUND-Kommando in BASIC, nur die Reihenfolge ist etwas vertauscht.

Eine Tonperiode von 0 erzeugt keinen Ton, sondern eine Pause.

Bei einer negativen Dauer wird die Länge der Volumen-Hüllkurve als Maß genommen. Diese wird dann –nmal abgespielt. Die Angabe 0 für die Dauer entspricht –1 (einmal die Hüllkurve abspielen).

Die SOUND QUEUE-Events (ON SQ GOSUB) aller Kanäle, die durch diesen Ton bedient wurden (Kanalstatus!), werden deaktiviert.

Die mit SOUND HOLD angehaltenen Töne werden wieder freigegeben.

BCAD SOUND CHECK Frage an, ob in der Ton-Warteschlange ein Platz für einen neuen Ton frei ist.

Eingaben: A = Bitmaske für den zu testenden Kanal
Ausgaben: A = Kanalstatus
Unverändert: IX, IY

Auch wenn bei der Eingabe mehrere Bits in A gesetzt sind, wird nur ein Kanal getestet, und zwar A vor B und B vor C.

Der Kanalstatus entspricht der Funktion SQ(n) in BASIC:

Bits 0,1,2 = Anzahl freier Plätze in der Queue des Kanals.
Bits 3,4,5 = Gesetzt, wenn der erste Ton der Queue ein Rendezvous mit Kanal A, B und/oder C hat.
Bit 6 = Erster Ton befindet sich im Haltezustand (Hold).
Bit 7 = Erster Ton ist gerade aktiv (kann nicht gleichzeitig mit Bit 6 gesetzt sein).

Das SOUND QUEUE-Event (ON SQ GOSUB) des getesteten Kanals wird deaktiviert.

BCB0 SOUND ARM EVENT Bestimme eine Routine, die aufgerufen werden soll, sobald in der Ton-Warteschlange ein Platz frei ist. (ON SQ GOSUB)

Eingaben: A = Bitmaske für Kanal
HL = Adresse des Event-Blocks
Ausgaben: keine
Unverändert: IX, IY

Auch wenn bei der Eingabe mehrere Bits in A gesetzt sind, wird nur ein Kanal aktiviert, und zwar A vor B und B vor C.

Der Event-Block muß vollständig initialisiert sein.

Der EVENT BLOCK wird automatisch deaktiviert, wenn

- er angestoßen wurde (KICK)
- SOUND QUEUE aufgerufen wird
- SOUND CHECK aufgerufen wird

BCB3 SOUND RELEASE Gib die Tonausgabe frei.

Eingaben: A = Bitmaske für die freizugebenden Kanäle
Ausgaben: keine
Unverändert: IY

Alle Töne, die durch SOUND HOLD eingefroren wurden, werden freigegeben. Zusätzlich werden die angegebenen Kanäle freigegeben, wenn diese durch das Hold-Bit im Kanalstatus festgehalten wurden.

BCB6 SOUND HOLD Friere die Tonausgabe ein.

Eingaben: keine
Ausgaben: CY = 1 → Es waren Töne aktiv, die nun angehalten wurden.
Unverändert: DE, IX, IY

Die Tonausgabe wird sofort unterbrochen. Ist CY = 0, so waren keine Töne aktiv.

Die Tonausgabe wird wieder fortgesetzt, wenn SOUND QUEUE, SOUND RELEASE oder SOUND CONTINUE aufgerufen wird.

BCB9 SOUND CONTINUE Setze die Tonausgabe fort.

Eingaben: keine
Ausgaben: keine
Unverändert: HL, IY

Die Tonunterbrechung durch SOUND HOLD wird beendet. Sind keine Töne angehalten worden, bewirkt die Routine nichts.

BCBC SOUND AMPL ENVELOPE Lege eine Amplituden-Hüllkurve neu fest.

Eingaben: A = Hüllkurvennummer
HL = Adresse des Parameterblocks
Ausgaben: CY = 1 → o.k., CY = 0 → ungültige Hüllkurvennummer
Unverändert: IX, IY, wenn CY = 0, dann auch HL, BC und A

Der Parameterblock der Amplituden-Hüllkurve besteht aus:

DEFB Anzahl Hüllkurvenabschnitte

danach entsprechend oft:

DEFB Anzahl Schritte (0 ... 127)
DEFB Schritthöhe (0 ... 255 modulo 16!)
DEFB Schrittlänge (0 ... 255. 0 = 256!)

Es können maximal fünf Abschnitte angegeben werden. Die Anzahl wird aber nicht überprüft! Gibt man null Abschnitte an, wird die "Standard-Hüllkurve" (zwei Sekunden lang ein konstanter Ton) eingestellt.

Der Block darf überall im RAM liegen, wenn er durch kein ROM verdeckt ist (also nicht im unteren Viertel).

Die erlaubten Hüllkurvennummern sind 1 bis 15. Die Funktion des Vektors und die Bedeutung der einzelnen Bytes im Parameterblock entsprechen dem Befehl ENV in BASIC.

Absolute Volumeneinstellungen werden mit der Schrittzahl 0 gekennzeichnet.

Ein Abschnitt kann auch eine Hardware-Hüllkurve des PSG definieren. Er muß dann wie folgt aufgebaut sein:

DEFB #80 + Wert für Register 13 (Hüllkurvenform)
DEFW Periodenlänge für den Hüllkurvengenerator (Reg. 11,12)

BCBF SOUND TONE ENVELOPE Lege eine Frequenz-Hüllkurve neu fest.

Eingaben:	A = Hüllkurvennummer
	HL = Adresse des Parameterblocks
Ausgaben:	CY = 1 → o.k., CY = 0 → ungültige Hüllkurvennummer
Unverändert:	IX, IY, wenn CY = 0, dann auch A, BC, HL

Die Funktion dieses Vektors und seine Parameter sind identisch mit dem Befehl ENT in BASIC.

DEFB Anzahl Hüllkurvenabschnitte (+ #80 für Repeat)

danach entsprechend oft:

DEFB Anzahl Schritte (0 ... &EF)
DEFB Schritthöhe (–128 ... +127)
DEFB Schrittlänge (0 ... 255. 0 = 256!)

Ist Bit 7 im ersten Byte (Anzahl Abschnitte) gesetzt, so wird diese Hüllkurve bei Bedarf beliebig oft wiederholt, bis der Ton abgearbeitet ist.

Abweichend davon können auch absolute Tonperiodenlängen (=APL) eingestellt werden. Ein Abschnitt ist dann wie folgt aufgebaut:

DEFB APL\256 OR &F0 ; MSB der neuen Tonperidenlänge.
DEFB APL AND &FF ; LSB der neuen Tonperidenlänge.
DEFB Schrittlänge

Achtung: Das Byte mit dem höherwertigen Nibble (Bits 8 bis 11) kommt zuerst! Außerdem müssen hier die Bits 4 bis 7 gesetzt sein, um diesen Abschnitt von einem normalen und relativen zu unterscheiden.

Ansonsten gelten die allgemeinen Angaben bei &BCBC SOUND AMPL ENVELOPE.

BCC2 SOUND A ADDRESS Erfrage die Adresse einer Amplituden-Hüllkurve.

Eingaben: A = Hüllkurvennummer
Ausgaben: CY = 1 → o.k.: HL = Adresse der Hüllkurve
BC = Länge der Hüllkurve (immer 16)
Unverändert: DE, IX, IY, wenn CY = 0, dann auch BC

Liegt die Nummer im Register A nicht im zulässigen Bereich (1 bis 15), kehrt die Routine mit CY=0 zurück.

BCC5 SOUND T ADDRESS Erfrage die Adresse einer Frequenz-Hüllkurve.

Eingaben: A = Hüllkurvennummer
Ausgaben: CY = 1 → o.k.: HL = Adresse der Hüllkurve und
BC = Länge der Hüllkurve (immer 16)
Unverändert: DE, IX, IY, wenn CY = 0, dann auch BC

Liegt die Nummer im Register A nicht im zulässigen Bereich (1 bis 15), kehrt die Routine mit CY=0 zurück.

THE KERNEL (KL) – Die Zentrale

BCC8 KL CHOKE OFF Setze den Kernel zurück.

Eingaben: keine
Ausgaben: B = ROM-Select-Adresse des laufenden Vordergrund-Programms
DE = Kaltstartadresse des laufenden Vordergrund-Programms
C = Flag: &FF → ROM- und &00 → RAM-Vordergrund-Programm
Unverändert: IX, IY

KL CHOKE OFF wird hauptsächlich von &BD13 MC BOOT PROGRAM benutzt, und dafür sind auch die Ausgabeparameter gedacht. Schlägt ein Ladevorgang fehl, orientiert sich MC BOOT PROGRAM an diesen Werten von KL CHOKE OFF, um zu einem lauffähigen Vordergrund-Programm zurückzukehren. Sind C und DE gleich Null, so erfolgte der Aufruf von einem anderen RAM-Vordergrund-Programm aus. MC BOOT PROGRAM kehrt im Fehlerfall dann zum Default-Vordergrund-ROM 0 (BASIC-ROM) zurück.

KL CHOKE OFF löscht die Synchronous Pending Queue und alle Ticker Chains bis auf die Events des Sound-Managers und der Tastaturabfrage. Außerdem wird der Interrupt verboten, was aber bei Aufruf über diesen Vektor (der mit einem Restart gebildet ist) nicht weiter interessiert, da dieser ja wieder Interrupts zuläßt.

BCCB KL ROM WALK Suche und initialisiere alle Hintergrund-ROMs.

Eingaben: DE = Adresse des 1. Bytes des freien Speicherbereichs (incl.)
HL = Adresse des letzten Bytes (incl.)
Ausgaben: DE = Adresse des 1. Bytes des freien Speicherbereichs (incl.)
HL = Adresse des letzten Bytes (incl.)
Unverändert: IX, IY

CPC 464: Es werden alle ROM-Nummern von 1 bis 7 abgearbeitet. Jedes Hintergrund-ROM (Kennung &01 auf Adresse &C000) wird initialisiert.

CPC 664 und 6128: Es werden alle ROM-Nummern von 0 bis 15 berücksichtigt. Vor der Initialisierung eines Hintergrund-ROMs wird noch getestet, ob es nicht vielleicht das laufende Vordergrund-ROM ist, das sich dann nicht wieder selbst initialisieren darf.

Das ist interessant, weil das AMSDOS-ROM, das als Hintergrund-ROM normalerweise die ROM-Select-Adresse 7 hat, nach Setzen einer einzigen Drahtbrücke auf der Platine zum Power-Up-ROM mit der ROM-Adresse 0 wird (und das BASIC-ROM auf dieser Adresse ausblendet). Durch eine entsprechende Programmierung der Initialisierungsroutine dieses ROMs wird der Schneider CPC damit zum CP/M-Rechner. Obwohl dieses ROM jetzt immer noch im ROM-Header als Hintergrund-ROM deklariert ist, ist es jetzt de facto das laufende Vordergrund-ROM.

Die ROMs werden initialisiert, indem &BCCE KL INIT BACK aufgerufen wird. Dort ist auch die Parameterübergabe beschrieben.

BCCE KL INIT BACK Initialisiere ein bestimmtes Hintergrund-ROM.

Eingaben:	C	= ROM-Selekt-Adresse des gewünschten ROMs
	DE	= Adresse des ersten Bytes des freien Speicherbereichs (incl.)
	HL	= Adresse des letzten Bytes (incl.)
Ausgaben:	DE	= Adresse des ersten Bytes des freien Speicherbereichs (incl.)
	HL	= Adresse des letzten Bytes (incl.)
Unverändert:	C, IX, IY	

CPC 464: Das gewünschte ROM wird nur initialisiert, wenn es sich auch um ein Hintergrund-ROM handelt und die ROM-Selekt-Adresse im Bereich 1 bis 7 liegt.

CPC 664 und 6128: Wie bei &BCCB KL ROM WALK sind jetzt zusätzlich ROMs im Bereich 0 bis 15 erlaubt. Ebenfalls wird nun vorher getestet, ob das zu initialisierende ROM nicht vielleicht das laufende Vordergrund-ROM ist.

Im Schneider CPC wird das frei verfügbare RAM dynamisch zugeteilt: Sobald der Rechner zurückgesetzt ist, wird das Vordergrund-Programm aufgerufen. Dabei übergibt der Kernel im HL- und DE-Register die Unter- und Obergrenze des frei verfügbaren RAMs. Das Vordergrund-Programm kann sich davon zunächst oben und unten einen Bereich für den eigenen Gebrauch reservieren (einfach, indem es die Registerwerte anpaßt) und sollte dann das oder die Hintergrund-ROMs initialisieren. Diese übernehmen nun ihrerseits die Grenzen, reservieren sich einen bestimmten Bereich und geben die neuen Grenzen zurück. Ist das letzte ROM initialisiert, erhält auch das Vordergrund-Programm in DE und HL den nun noch verbliebenen freien RAM-Bereich zurück.

Es ist zu beachten, daß Hintergrund-ROMs nicht erwarten können, daß sie ihren RAM-Bereich immer an derselben Stelle zugeteilt bekommen! Als Erleichterung wird aber unter anderem beim &0018 FAR CALL (RST 3) in ein bestimmtes ROM immer das IY-Register auf den Wert plus 1 gesetzt, den die Initialisierungsroutine des ROMs im HL-Register zurückgegeben hatte. Alle ROMs müssen einen ROM-Header haben, der die ersten Bytes beansprucht. Dieser Header ist wie folgt aufgebaut:

```
&C000 – DEFB ROMTYP – Vorder-Hintergrund-Erweiterungs-ROM
&C001 – DEFB MARKNR – Mark Number          }
&C002 – DEFB VERSION – Version Number      } Statistics
&C003 – DEFB MODIFI – Modification Level   }
&C004 – ff          – External Command Table
```

Der erste Eintrag in der Tabelle externer Kommandos muß die Initialisierungsroutine des entsprechenden ROMs sein. Der Aufbau dieser Tabelle ist bei &BCD1 KL LOG EXT beschrieben.

BCD1 KL LOG EXT Mache dem Kernel eine RSX-Erweiterung bekannt.

Eingaben: BC zeigt auf Kommandotabelle
HL zeigt auf 4 Bytes RAM für den Kernel
Ausgaben: keine
Unverändert: AF, BC, HL, IX, IY

Resident System Extensions (RSX) werden vom Hauptprogramm meist erst ins RAM nachgeladen und dann initialisiert. Solche Zusatzprogramme müssen ihre eigene Lage und die Lage ihrer Datenfelder mit dem Hauptprogramm selbst abstimmen.

Es ist dabei Aufgabe dieser Zusatzprogramme, sich selbst dem Kernel vorzustellen. Unterbleibt dies, kann das Hauptprogramm nur bei genauer Kenntnis aller Einsprungadressen auf die jeweiligen Routinen zugreifen.

Nach der Einbindung als RSX (mit diesem Vektor) sind diese Routinen aber dem Kernel mit Namen bekannt, was einen Lage-unabhängigen Zeiger auf sie ermöglicht.

Sowohl die Kommandotabelle als auch die freien 4 Bytes müssen im zentralen RAM liegen (&4000 bis &BFFF). Die Kommandotabellen der ROMs müssen an der angegebenen Stelle im ROM selbst liegen.

Die Kommandotabelle besteht aus zwei Teilen:

1. Jumpblock

```
DEFW NAMTAB     ;  Zeiger auf die Namenstabelle
JP NAME1        ;  Vektor zum ersten Eintrag
JP NAME2        ;  Vektor zum zweiten Eintrag
...
```

2. Namenstabelle (NAMTAB)

```
DEFM "name 1"
DEFM "name 2"
...
DEFB 0
```

Der Jumpblock muß so viele Vektoren umfassen, wie in der Namenstabelle Namen eingetragen sind. Die Vektoren können auch mit Restarts gebildet sein.

Jeder Eintrag in der Namenstabelle kann bis zu 16 Buchstaben lang sein. Der letzte Buchstabe muß dabei jeweils mit &80 geodert sein (Bit 7 gesetzt). Abgeschlossen wird diese Tabelle mit einem Null-Byte.

BCD4 KL FIND COMAND Erfrage zum angegebenen Namen einer RSX oder eines ROMs die Ausführungsadresse und Speicherkonfiguration.

Eingaben: HL zeigt auf den Kommandonamen. Im letzten Buchstabe des Namens muß Bit 7 gesetzt sein.
Ausgaben: CY = 0 → nicht gefunden
CY = 1 → gefunden, HL = Adresse und
C = ROM-Selekt-Byte
Unverändert: IX, IY

Die zurückgegebenen Werte sind geeignet, um mit &001B KL FAR PCHL direkt die Routine aufrufen zu können. Benötigt man aber auch das C- oder HL-Register für die Parameterübergabe, so muß man, erst eine Far Address um dann &0018 KL FAR CALL (RST 3) aufzurufen.

Diese Routine durchsucht alle Kommandotabellen, ob sie nun von einer RSX stammen oder in einem ROM liegen. Dabei werden die zuletzt angefügten Tabellen zuerst durchsucht. Das ist wichtig, da bei gleichbenannten Routinen immer nur die letzte gefunden wird. Die andere wird "verdeckt". Die ROMs werden von ROM-Nummer 0 an aufwärts bis zur ersten unbenutzten ROM-Nummer über 7 (CPC 464) bzw. 15 (CPC 664 oder 6128) durchsucht.

Wird das Kommando in einem Vordergrund-ROM gefunden, so wird ein Kaltstart dieses Programms durchgeführt. Dann kehrt KL FIND COMMAND nicht mehr zurück.

BCD7 KL NEW FRAME FLY Initialisiere einen Datenblock, und füge ihn in die Liste aller Software-Unterbrechungen bei jedem Strahlrücklauf des Bildschirms ein.

Eingaben: HL zeigt auf den Frame Flyback Block
B = Event-Klasse (express, Priorität o.ä.)
C = ROM-Selekt-Byte für die EVENT-Routine
DE = Adresse der Event-Routine

Ausgaben: keine
Unverändert: BC, IX, IY

Der mit HL angezeigte Block wird entsprechend B, C und DE initialisiert, der Kick Counter auf 0 eingestellt und der Block in die Frame Flyback Chain eingehängt, falls er nicht schon eingehängt ist. Die angegebene Routine wird nun mit jedem Strahlhochlauf im Monitorbild aufgerufen. Der Frame Flyback Block muß im zentralen RAM-Bereich liegen und besteht aus zwei freien Bytes für den Kernel (Kettungs-Pointer) und einem Event-Block (7 Bytes lang).

BCDA KL ADD FRAME FLY Füge einen fertigen Datenblock in die Liste aller Software-Unterbrechungen bei jedem Strahlrücklauf ein.

Eingaben: HL zeigt auf den Frame Flyback Block
Ausgaben: keine
Unverändert: BC, IX, IY

Der mit HL angezeigte Block muß fertig initialisiert sein und wird vom Kernel in die Frame Flyback Chain eingehängt, wenn er sich noch nicht darin befindet. Dieser Block muß im zentralen RAM-Bereich liegen und besteht aus zwei Bytes für den Kernel (Kettungs-Pointer) und einem Event-Block.

BCDD KL DEL FRAME FLY Entferne einen Datenblock aus dieser Liste.

Eingaben: HL zeigt auf den Frame Fly Block
Ausgaben: keine
Unverändert: BC, IX, IY

Wenn der angezeigte Block in der Frame Flyback-Liste ist, wird er ausgehängt. Dadurch wird dieser Block nicht mehr angestoßen. Handelt es sich um ein synchrones Event, so können aber noch einige Interrupts ausstehen, die noch abgearbeitet werden. Um auch das zu verhindern, muß man auch noch &BCF8 KL DEL SYNCHRONOUS aufrufen.

BCE0 KL NEW FAST TICKER Initialisiere einen Datenblock und füge ihn in die Liste aller Software-Unterbrechung bei jedem Interrupt ein.

Eingaben: HL zeigt auf den Fast Ticker Block
B = Event-Klasse (asynchron, Priorität o.ä.)

Eingaben: C = ROM-Selekt-Byte für die Event-Routine
DE = Adresse der Event-Routine
Ausgaben: keine
Unverändert: BC, IX, IY

Der durch HL angezeigte Block wird entsprechend B, C und DE initialisiert, der Kick Counter auf 0 eingestellt und der Block in die Fast Ticker Chain eingehängt, wenn er sich nicht bereits in ihr befindet. Damit wird mit jedem Interrupt (300mal pro Sekunde) die entsprechende Routine aufgerufen. Der Fast Ticker Block muß im zentralen RAM liegen und besteht aus zwei freien Bytes für den Kernel (Kettungs-Pointer) und einem Event-Block.

BCE3 KL ADD FAST TICKER Füge einen fertigen Datenblock in die Liste aller Software-Unterbrechungen bei jedem Interrupt ein.

Eingaben: HL zeigt auf den Fast Ticker Block
Ausgaben: keine
Unverändert: BC, IX, IY

Wie &BCE0 KL NEW FAST TICKER, nur daß der Block bereits vollständig initialisiert sein muß.

BCE6 KL DEL FAST TICKER Entferne einen Datenblock aus dieser Liste.

Eingaben: HL zeigt auf den Fast Ticker Block
Ausgaben: keine
Unverändert: BC, IX, IY

Der durch HL angezeigt Block wird aus der Fast Ticker Chain wieder ausgehängt, wenn er sich noch darin befindet. Dadurch wird die entsprechende Routine nicht mehr angestoßen. Bei synchronen Events können aber noch Kicks ausstehen, die dann noch abgearbeitet würden. Um das zu verhindern, muß man auch noch &BCF8 KL DEL SYNCHRONOUS aufrufen.

BCE9 KL ADD TICKER Füge einen Datenblock in die Liste des Universal-Timers ein.

Eingaben: HL zeigt auf den Ticker Block
DE = Startverzögerung (Count Down)
BC = Wiederholverzögerung (Reload Count)

Ausgaben: keine
Unverändert: BC, IX, IY

Der durch HL angezeigte Block wird in die Ticker Chain eingehängt, wenn er sich nicht bereits darin befindet. Der im Tickerblock enthaltene Event-Block muß vollständig initialisiert sein. Der Tickerblock muß sich im zentralen RAM befinden und ist wie folgt aufgebaut:

- 2 Bytes Platz für den Kernel (Kettungs-Pointer)
- 2 Bytes Count Down
- 2 Bytes Reload Count
- 7 Bytes Event-Block

Der normale Ticker ist der universellste von allen Software-Timern: 50mal in der Sekunde wird diese Liste abgearbeitet. Dabei wird zunächst nur der Count Down-Wert heruntergezählt und dann erst die Routine angestoßen. Mit jedem Kick wird dann der Reload-Wert in den Count Down geladen und dieser dann von neuem heruntergezählt. Ein Nachladewert 0 blockiert jedoch weitere Kicks. Der Block verbleibt zwar in der Liste (und verbraucht nach wie vor Rechenzeit), wird aber nicht mehr angestoßen. Mit Hilfe dieses Mechanismus werden EVERY und AFTER in BASIC realisiert.

BCEC KL DEL TICKER Entferne einen Datenblock aus der Liste des Universal-Timers.

Eingaben: HL zeigt auf den Ticker Block
Ausgaben: CY = 1 → Der Block war in der Liste
DE enthält den verbliebenen Wert aus dem Count Down
CY = 0 → Der Block war nicht in der Liste
Unverändert: BC, IX, IY

Der mit HL adressierte Block wird aus der Ticker Chain entfernt, wenn er in ihr eingehängt war. Dann enthält DE den Rest aus dem Count Down (entspricht dem BASIC-Befehl REMAIN). Dadurch wird dieser Block nicht mehr angestoßen. Bei synchronen Events (wie BASIC sie benutzt) werden die ausstehenden Kicks aber noch abgearbeitet. Soll auch das unterbunden werden, muß man noch &BCF8 KL DEL SYNCHRONOUS aufrufen.

BCEF KL INIT EVENT Beschreibe einen Event-Block vorschriftsmäßig mit den in Registern angegebenen Werten.

Eingaben: HL zeigt auf den Event-Block
B = Event-Klasse (asynchron, Priorität o.ä.)
C = ROM-Selekt-Byte für die Event-Routine
DE = Adresse der Event-Routine
Ausgaben: HL zeigt hinter den Event-Block, wurde also um 7 erhöht
Unverändert: AF, BC, DE, IX, IY

Der Event-Block wird mit den in B, C und DE angegebenen Werten initialisiert, und der Kick Counter wird auf 0 eingestellt. Der Block muß im zentralen RAM-Bereich (&4000 bis &BFFF) liegen, damit der Kernel vom Interrupt-Pfad aus jederzeit darauf zugreifen kann. Der Event-Block ist folgendermaßen aufgebaut:

- 2 Bytes Platz für den Kernel (Kettungs-Pointer)
- 1 Byte Kick-Counter (Zähl- und Steuerbyte)
- B: 1 Byte Class (Event-Art: express/asynchron/Priorität...)
- DE:2 Byte Adresse der Event-Behandlungsroutine
- C: 1 Byte ROM-Select-Byte (außer, wenn Near Address angegeben)

Der Event-Block ist normalerweise Teil eines Tickers, bzw. Fast Tickers oder Frame Flyback Blocks. Auch für die Sound-Interrupts und das Break-Event muß so ein Event-Block eingerichtet werden. Bei Bedarf können auch External Interrupts mit Hilfe des Software-Interrupt-Mechanismus behandelt werden.

Man muß dabei genau zwischen den Interrupt-Quellarten (die gerade aufgelistet wurden) und den Event-Arten unterscheiden. Ist ein Event nämlich erst einmal angestoßen, sieht man nicht mehr, woher der Kick kam. Die Event-Art wird dabei in Byte 3 (Class) angegeben:

Bit 0 = 1 – Near Address (→ ROM-Selekt-Byte bedeutungslos)
Bit 1 – 4 – Priorität (nur bei synchronen Events)
Bit 5 – muß auf 0 gesetzt sein.
Bit 6 = 1 – Express Event
Bit 7 = 1 – Asynchronous Event (Bits 1 – 4 ohne Bedeutung)

Anmerkung: Die Bits 1 bis 6 werden benutzt, um eine Art "Gesamt-Priorität" zu bilden. Durch ein gesetztes Bit 6 erhalten alle Express Events automatisch eine höhere Priorität als alle "normalen". Bit 5 wird benutzt, um die Funktion der Vektoren &BD04 KL EVENT DISABLE und &BD07 KL EVENT ENABLE zu realisieren. Durch ein gesetztes Bit 5 werden ebenfalls alle "normalen" Events übertrumpft.

BCF2 KL EVENT Stoße einen Event-Block an. Dies führt zu einer Software-Unterbrechung.

Eingaben: HL zeigt auf den Event-Block
Ausgaben: keine
Unverändert: IX, IY

Diese Routine ist dafür vorgesehen, vom Interrupt-Pfad aus aufgerufen zu werden! Speziell ist hierbei an den External Interrupt gedacht, um auch solchen Unterbrechungen den vollen Service des Event-Mechanismus bereitzustellen. Man kann sie zwar auch von einem normalen Programm aus aufrufen, doch ist das nur in Ausnahmefällen sinnvoll.

Weil der für den Vektor verwendete Restart (RST 1 LOW JUMP) Interrupts wieder zulassen würde, kann diese Routine vom Interrupt-Pfad aus nicht über den Vektor aufgerufen werden. Dann muß man sich die Adresse aus dem Vektor holen (Bits 14 und 15 der Adresse zurücksetzen!) und die Routine direkt anspringen.

BCF5 KL SYNC RESET Lösche die Liste aller synchronisierbaren Unterbrechungen, die noch auf ihre Ausführung warten.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Alle Events, die noch in der Synchronous Event Pending Queue eingereiht sind und auf ihre Ausführung warten, werden einfach vergessen. Die aktuelle Event-Priorität wird wieder auf 0 gesetzt.

In den so ausgehängten Event-Blocks werden keine Änderungen vorgenommen. Die Queue (eine verkettete Liste) wird geleert, indem ihr Anfangszeiger auf 0 gestellt wird.

Deshalb wird auch das Byte 2 (Kick Counter) der Event-Blocks nicht wieder auf 0 gestellt. Wird ein solcher Event-Block später wieder angestoßen, wird dann nur noch der Kick Counter erhöht, der Event-Block aber nicht in die Pending Queue eingehängt (der Kernel nimmt nämlich an, daß der Block sich bereits darin befindet, weil er noch ein paar Kicks ausstehen hat). Der Block ist de facto ruhiggestellt. Das laufende Programm muß selbst sicherstellen, daß die Kick Counter aller betroffenen Event-Blocks auf 0 zurückgestellt werden.

BCF8 KL DEL SYNCHRONOUS Stelle den Event-Block einer synchronisierbaren Unterbrechung ruhig und entferne ihn auch aus der Warteliste, falls er dort eingetragen ist.

Eingaben: HL zeigt auf den Event-Block
Ausgaben: keine
Unverändert: IX, IY

Der durch HL angezeigte Event-Block wird ruhiggestellt und, falls nötig, aus der Synchronous Event Pending Queue entfernt. Dadurch werden alle noch ausstehenden Kicks vergessen.

Um eine synchrone Interrupt-Quelle abzustellen, muß man erst den entsprechenden Block aus der Ticker, Fast Ticker oder Frame Flyback Chain entfernen, damit keine Kicks mehr erzeugt werden, und dann die ausstehenden Kicks mit diesem Vektor "wegwerfen". Die restlichen Interrupt-Quellen (External Interrupt, BREAK oder SOUND) müssen ähnlich behandelt werden.

BCFB KL NEXT SYNC Hole die nächste synchronisierbare Unterbrechung aus der Warteschlange, falls nocheine wartet.

Eingaben: keine
Ausgaben: CY = 0 → Die Synchronous Event Pending Queue ist leer
CY = 1 → Es war ein Block in der Queue vorhanden
HL zeigt auf den Event-Block
A = alte Event-Priorität
Unverändert: BC, IX, IY

War ein Block in der Synchronous Event Pending Queue vorhanden, so wird er aus dieser entfernt. Außerdem wird die aktuelle Event-Priorität dem neuen Event angepaßt. Dadurch kann die Warteschlange auch gepollt werden, während ein synchrones Event abgearbeitet wird.

Events mit niedrigerer oder gleicher Priorität werden dann vom Kernel einfach versteckt, wichtigere Unterbrechungen werden aber weitergemeldet und können so die laufende Event-Routine noch einmal unterbrechen.

Nun muß &BCFE KL DO SYNC aufgerufen werden! Dieser Vektor übernimmt die Blockadresse aus dem HL-Register und ruft die dort eingetragene Event-Behandlungsroutine auf.

Danach muß &BD01 KL DONE SYNC aufgerufen werden, wofür alle Ausgabewerte von KL NEXT SYNC benötigt werden. KL DONE SYNC vermindert den Kick Count und fügt den Event-Block, falls nötig, wieder in die Pending Queue ein. Zum Schluß wird wieder die alte Event-Priorität eingestellt.

Diese drei Funktionen können normalerweise nur in folgendem Zusammenspiel aufgerufen werden:

```
        CALL KL NEXT SYNC
        JR   NC,WEITER
        PUSH AF
        PUSH HL
        CALL KL DO SYNC
        POP  HL
        POP  AF
        CALL KL DONE SYNC
WEITER: ...
```

Variationsmöglichkeiten ergeben sich hauptsächlich nur durch den Einbau einer Schleife (Pollen, bis die Queue leer ist) und durch &B921 HI KL POLL SYNCHRONOUS.

Diese Routine testet nur, ob in der Pending Queue ein Block wartet und ist dabei erheblich schneller als KL NEXT SYNC, weil sie im RAM liegt und nicht über einen Restart angesprungen werden muß.

BCFE KL DO SYNC Führe die mit &BCFB geholte Unterbrechung aus.

Eingaben: HL zeigt auf den Event-Block
Ausgaben: keine
Unverändert: IX, IY

Die Verwendung dieses Vektors ist bei &BCFB KL NEXT SYNC beschrieben.

BD01 KL DONE SYNC Muß nach &BCFE aufgerufen werden.

Eingaben: HL zeigt auf den Event-Block
A ist die alte Event-Priorität
Ausgaben: keine
Unverändert: IX, IY

Dieser Vektor ist ebenfalls bei &BCFB KL NEXT SYNC beschrieben.

BD04 KL EVENT DISABLE Lege die Ausführung von normalen synchronisierbaren Unterbrechungen auf Eis.

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, IX, IY

Wie bei &BCEF KL INIT EVENT erwähnt, bilden die Bits 1 bis 6 im Art-Byte (Class) eines Event-Blocks eine "Gesamt-Priorität". Bit 5 soll dabei immer auf Null gesetzt werden. Der Kernel hat im RAM eine Speicherstelle, in der er entsprechend die Priorität des momentan bearbeiteten Events speichert (oder 0, falls zur Zeit kein Software-Interrupt bearbeitet wird). KL EVENT DISABLE setzt Bit 5 in diesem Byte auf Eins. Dadurch ergibt sich eine "Gesamt-Priorität", die höher ist, als die höchste Priorität eines "normalen" synchronen Events. Beim Pollen der Synchronous Event Pending Queue werden deshalb alle normalen Events verborgen, gerade so, als ob momentan ein Event mit der Priorität &X10000 bearbeitet würde.

Eigentlich nicht vorgesehen aber durchaus möglich ist es, daß ein express-synchrones Event KL EVENT DISABLE aufruft. Dadurch wird dann auch die Behandlung aller Express-Events ausgesetzt, weil jetzt sowohl Bit 6 als auch Bit 5 gesetzt sind.

Wird KL EVENT DISABLE oder auch &BD07 KL EVENT ENABLE von einer Event-Behandlungsroutine aus aufgerufen, bleibt der Zustand von Bit 5 nicht erhalten, wenn diese Routine mit RET abgeschlossen wird. Dann wird nämlich mittels &BD01 KL DONE SYNC die alte Priorität wieder restauriert.

Die Routine &BD07 KL EVENT ENABLE hebt den Effekt von KL EVENT DISABLE wieder auf, indem sie Bit 5 zurücksetzt.

BASIC verwendet nur "normale", synchrone Events für seine Interrupt-Mechanismen. Die BASIC-Befehle DI und EI rufen KL EVENT DISABLE bzw. &BD07 KL EVENT ENABLE auf. Dadurch werden alle Events vom Kernel verborgen, wenn der BASIC-Interpreter zwischen zwei Statements pollt.

Einzige Ausnahme bildet dabei die BREAK-Behandlung: Hierfür wird ein express-synchrones Event benutzt. Deshalb kann man auch nach DI noch breaken. Auch ON BREAK GOSUB wird dadurch nicht beeinflußt.

BD07 KL EVENT ENABLE Lasse die Ausführung von normalen synchronisierbaren Unterbrechungen wieder zu.

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, IX, IY

Dieser Vektor ist mit &BD04 KL EVENT DISABLE zusammen beschrieben.

BD0A KL DISARM EVENT Stelle einen Event-Block ruhig.

Eingaben: HL zeigt auf den Event-Block
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Der Event-Block wird dadurch ruhiggestellt, daß der Kick-Zähler auf einen negativen Wert (–64) gesetzt wird. Noch ausstehende Kick-Behandlungen gehen dadurch verloren. Diese Routine ist nur für asynchrone Event-Block gedacht.

Für synchrone Events soll man &BCF8 KL DEL SYNCHRONOUS benutzen, da sich ein solches Event gerade in der Synchronous Event Pending Queue befinden könnte (weil es noch auf eine Kick-Bearbeitung wartet). Wird es dort nicht ausgehängt, so wird es noch einmal aufgerufen.

Der Ticker, Fast Ticker oder Frame Flyback Block, von dem der Event-Block eventuell ein Bestandteil ist, wird dadurch nicht aus seiner Liste entfernt. Eintreffende Kicks werden in Zukunft einfach ignoriert.

BD0D KL TIME PLEASE Erfrage den Stand des Interrupt-Zählers.

Eingaben: keine
Ausgaben: DEHL enthält die Zeit in 1/300stel Sekunden
Unverändert: AF, BC, IX, IY

Für allgemeine Zeitmessungen lohnt es sich oft nicht, den CPC mit der Rechenzeit für einen Event-Block zu belasten. Verbrauchte Zeit läßt sich mit diesem Vektor bequem feststellen. In BASIC wird dieser Vektor für die reservierte Variable TIME benutzt. Da dieser Zähler bei jedem Hardware-Interrupt weitergestellt wird, leidet seine Genauigkeit mit jedem Kassetten- oder Diskettenzugriff.

BD10 KL TIME SET Stelle den Interrupt-Zähler auf einen neuen Startwert.

Eingaben: DEHL enthält den neuen Wert für den Interrupt-Zähler
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Den Zähler kann man auch stellen, um ihn beispielsweise an die Datums- und Uhrzeiteingabe in einem Programm anzupassen. Der gesamte Zeit-Meßumfang des 4-Byte-Zählers umfaßt etwa 165 Tage.

BD5B KL RAM SELECT Wähle eine neue RAM-Konfiguration aus (nur CPC 6128).

Eingaben: neue RAM-Konfiguration
Ausgaben: alte RAM-Konfiguration
Unverändert: BC, DE, HL, IX, IY

Vom PAL, das für die RAM-Auswahl zuständig ist, werden nur die Bits 0, 1 und 2 ausgewertet.

Die ROM-Konfiguration (ROM-Status und ROM-Auswahl) bleibt davon unberührt, eingeblendete ROMs haben auch vor dem zusätzlichen RAM im CPC 6128 Priorität.

Das Video-RAM kann sich nur in einem Block der "normalen" 64-kByte-RAM befinden. Im Extremfall kann die CPU nicht auf den Bildschirmspeicher zugreifen, weil dieser sich in einem ausgeblendeten RAM-Block befindet.

Folgende Einstellungen sind möglich:

CPU-Adreßviertel				Konfiguration wird	normale Lage
A –0–	–1–	–2–	–3–	verwendet bei:	für Video-RAM
0	n0	n1	n2	n3	Normalzustand	n3=Screen
1	n0	n1	n2	z3	CP/M: BIOS, BDOS etc.	n1=Screen
2	z0	z1	z2	z3	CP/M: TPA	(n1=Screen)
3	n0	n3	n2	z3	CP/M: n3=Hush-Tabelle	(n1=Screen)
4	n0	z0	n2	n3	Bankmanager	n3=Screen
5	n0	z1	n2	n3	Bankmanager	n3=Screen
6	n0	z2	n2	n3	Bankmanager	n3=Screen
7	n0	z3	n2	n3	Bankmanager	n3=Screen

n = normale RAM-Bank z = zusätzliches RAM

THE MACHINE PACK (MC)
Maschinennahe Routinen

BD13 MC BOOT PROGRAM Lade und starte ein Maschinencode-Programm.

Eingaben: HL = Adresse einer Laderoutine
Ausgaben: –/–
Unverändert: –/–

Diese Routine kann benutzt werden, um von einem laufenden Vordergrund-Programm aus (z.B. BASIC) ein Maschinencode-Programm von Kassette oder Diskette zu laden.

Das Betriebssystem initialisiert zuerst einen Teil der Firmware, damit von hier aus keine Störungen des Ladevorgangs auftreten können:

- Zurücksetzen von Peripheriegeräten
- Löschen aller Software-Interrupts (auch Sound)
- Default-Setzen der Indirections und
- Initialisieren des Maschinenstapels

ROM-Auswahl und -Status werden nicht geändert. Aber Achtung: Dieser Vektor ist mit einem LOW JUMP gebildet, der das obere ROM ausblendet (eventuell Routine aus dem Vektor herausholen).

Danach wird die in HL angegebene Routine aufgerufen, um das eigentliche Programm zu laden. Sie muß in der momentanen ROM-Konfiguration direkt erreichbar sein und folgende Aussprungbedingung erfüllen:

- CY = 0 → Ladefehler
- CY = 1 → o.k. und HL enthält die Einsprungsadresse des Programms

Wurde das Programm korrekt geladen (CY = 1), so wird das Betriebssystem komplett neu initialisiert. Andernfalls wird eine Fehlermeldung ausgegeben und in aller Regel zu BASIC zurückgekehrt.

BD16 MC START PROGRAM Rufe ein Vordergrund-ROM auf.

Eingaben: HL = Einsprungsadresse
C = ROM-Selekt-Byte
Ausgaben: –/–
Unverändert: –/–

Diese Routine kann benutzt werden, um ein Maschinenprogramm in einem Vordergrund-ROM oder im RAM (wenn dieses bereits geladen ist) zu starten.

Das Betriebssystem wird komplett zurückgesetzt und die übergebene Adresse mit einem FAR CALL angesprungen. C muß also entweder die entsprechende ROM-Nummer enthalten, mit der das benötigte ROM eingeblendet wird, oder den von einem RAM-Programm benötigten ROM-Status (&FC bis &FF).

BD19 MC WAIT FLYBACK Warte bis zum nächsten Strahlrücklauf des Monitorbildes.

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Mit jedem Strahlhochlauf des Monitorbildes erzeugt der CRTC ein Signal (VSYNC), das einerseits die Interrupt-Erzeugung des Gate Arrays synchronisiert (und einen Interrupt auslöst) und andererseits über Bit 0 von Port B der PIO eingelesen werden kann.

Die Zeit, in der der Elektronenstrahl des Monitors vom Bildschirmende wieder zu dessen Anfang hochläuft, ist vergleichsweise lang und damit ideal geeignet, umfangreichere Aktionen auf dem Bildschirm durchzuführen, ohne daß es eventuell zu unangenehmen Flimmereffekten kommt.

Dieser Vektor läßt die CPU so lange im Leerlauf kreisen, bis das Ende eines Bildes auf dem Monitor erkannt wird. Wenn möglich, sollte man hierfür aber immer ein Frame Flyback Event programmieren.

BD1C MC SET MODE Lege den Bildschirmmodus neu fest.

Eingaben: A = gewünschter Bildschirm-Modus
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Der in A übergebene Wert wird geprüft, und nur wenn er im erlaubten Bereich von 0 bis 2 ist, wird der Bildschirmmodus entsprechend geändert.

MC SET MODE informiert das Screen Pack nicht über den neuen Bildschirmmodus! Im Normalfall sollte man deshalb &BC0E SCR SET MODE benutzen.

BD1F MC SCREEN OFFSET Setze den Hardware-Scroll-Offset.

Eingaben: HL = neuer Scroll-Offset
A = neues Bildschirmviertel
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Der Bildwiederholspeicher wird entsprechend A in ein neues RAM-Viertel gelegt. Um einen gültigen Wert zu erhalten, wird A mit &C0 maskiert.

Außerdem wird der Hardware-Scroll-Offset entsprechend HL geändert. HL wird ebenfalls maskiert, um einen gültigen Wert zu erhalten, und zwar mit &07FE.

MC SCREEN OFFSET informiert das Screen Pack ebenfalls nicht! Im Normalfall sollte deshalb &BC05 SCR SET OFFSET oder &BC08 SCR SET BASE benutzt werden.

BD22 MC CLEAR INKS Setze alle Tinten auf eine Farbe.

Eingaben: DE zeigt auf eine Farbtabelle
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Diese Routine wird benutzt, um beim Löschen des Bildschirms (CLS) oder Screen-Mode-Wechsel (MODE) den Eindruck zu erwecken, daß der Bildschirm ohne Verzögerung gelöscht wurde (tatsächlich dauert es aber etwa eine Achtelsekunde, bis die 16000 Bytes des Video-RAMs gelöscht sind).

Die Farbtabelle besteht nur aus zwei Bytes:
Byte 0 = Paletten-Farbnummer für den Bildschirmrand (BORDER)
Byte 1 = Paletten-Farbnummer für alle Tinten (INKs)

Beim Einsatz dieser Routine muß man aber immer daran denken, daß das Screen Pack normalerweise auf dem Frame Flyback Ticker einen Event-Block eingehängt hat, durch den im SPEED-INK-Rhythmus für alle Tinten (INKs) eine der beiden programmierten Farben in das Gate Array geschrieben wird.

BD25 MC SET INKS Lege die Farben für alle Tinten neu fest.

Eingaben: DE zeigt auf eine Farbtabelle
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Dieser Vektor bildet das Pendant zu &BD22 MC CLEAR INKS. Hiermit kann man auf einen Schlag allen Tinten und dem Border neue Farben zuordnen. Die Farbtabelle hat dabei folgendes Layout:

Byte 0 – Paletten-Farbnummer für den Bildschirmrand (BORDER)
Byte 1 – Paletten-Farbnummer für Tinte (INK) 0
Byte 2 – Paletten-Farbnummer für Tinte (INK) 1
...
Byte 16 – Paletten-Farbnummer für Tinte 15

Dieser Vektor programmiert alle 16 Tinten, unabhängig vom aktuellen Bildschirmmodus. Trotzdem genügt es, nur so viele Bytes korrekt zu besetzen, wie momentan notwendig sind. Die restlichen INKs werden dann zwar auch (unvorhersehbar) verändert. Das macht aber nichts, weil sie ja nicht dargestellt werden können.

Auch hier gilt wieder zu bedenken, daß normalerweise die Farben für alle Tinten vom Screen Pack im SPEED-INK-Rhythmus ständig neu programmiert werden, wofür sogar genau diese Routine benutzt wird.

BD28 MC RESET PRINTER Setze die Printer-Indirections zurück.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Die Indirection &BDF1 IND MC WAIT CHAR wird auf die Standardroutine umgebogen.

Außerdem wird beim CPC 664 und 6128 die Drucker-Übersetzungstabelle mit ihren Standardwerten gefüllt.

BD2B MC PRINT CHAR Versuche, ein Zeichen zum Drucker zu schicken.

Eingaben: A = zu druckendes Zeichen
Ausgaben: CY = 1 → o.k.
CY = 0 → nicht gedruckt (time out)
Unverändert: BC, DE, HL, IX, IY

Dieser Vektor ruft &BDF1 IND MC WAIT CHAR auf, um ein Zeichen zu drucken. Dabei wird normalerweise Bit 7 von A ignoriert, weil der Centro-

nics-Anschluß am Schneider CPC nur 7 Datenleitungen besitzt. Die Indirection wartet, bis der Drucker nicht mehr BUSY signalisiert und druckt das Zeichen. Spätestens aber nach ca. 0,4 Sekunden kehrt die Routine zurück, damit das druckende Programm auf BREAK o.ä. testen kann.

Beim CPC 664 und 6128 werden alle Zeichen, die gedruckt werden sollen, vorher in der Printer Translation Table gesucht und eventuell geändert, bevor &BDF1 IND MC PRINT CHAR gerufen wird.

BD2E MC BUSY PRINTER Schaue nach, ob der Drucker bereit ist.

Eingaben: keine
Ausgaben: CY = 1 → Der Drucker ist BUSY/OFF LINE oder nicht angeschlossen.
CY = 0 → Der Drucker ist bereit, ein Zeichen zu empfangen.
Unverändert: A, BC, DE, HL, IX, IY

Vor einer Textausgabe kann man mit diesem Vektor sicherstellen, daß überhaupt ein Drucker angeschlossen ist. Im Zusammenspiel mit &BD31 MC SEND PRINTER kann wie mit &BD2B MC PRINT CHAR ein Zeichen zum Drucker geschickt werden.

BD31 MC SEND PRINT Sende ein Zeichen zum Drucker.

Eingaben: A = zu druckendes Zeichen
Ausgaben: CY = 1
Unverändert: BC, DE, HL, IX, IY

Dieser Vektor darf man nie aufrufen, ohne vorher den Status der BUSY-Leitung vom Drucker zu testen, da sonst die meisten übertragenen Zeichen verlorengehen.

BD34 MC SOUND REGISTER Lade ein Byte in ein Register des Sound- Chips.

Eingaben: A = Registernummer
C = Datenbyte für das Register
Ausgaben: keine
Unverändert: DE, HL, IX, IY

Im Normalfall treibt man den Tongenerator über den Sound-Manager, der eine sehr komfortable Programmierung der drei Tonkanäle erlaubt.

Will man aber trotzdem einzelne Register direkt programmieren, beispielsweise weil der Sound Manager für spezielle Effekte zu langsam ist, sollte man sich an diesen Vektor halten.

Nur in Ausnahmefällen, z.B. bei Sprach- oder Geräusch-Reproduktionen, die extrem zeitkritisch sind, wird man den Sound-Generator "zu Fuß" programmieren, da dies wegen der Hardware-Gestaltung im Schneider CPC eine ziemlich aufwendige Angelegenheit ist.

BD58 MC PRINT TRANSLATION Ändere die Zeichen-Übersetzungstabelle für den Drucker (nur CPC 664 und 6128).

Eingaben: HL zeigt auf die neue Übersetzungstabelle
Ausgaben: CY = 1 → o.k.
CY = 0 → Fehler (Tabelle zu lang)
Unverändert: IX, IY

Die Übersetzungstabelle hat folgenden Aufbau:

1. Byte – Anzahl der Einträge (maximal 20 Stück); danach jeweils 2 Bytes pro Eintrag

Jeder Eintrag besteht aus:

1. Byte – zu übersetzender Zeichencode
2. Byte – zugeordneter Zeichencode (&FF → Zeichen ignorieren)

Die Tabelle wird von MC PRINT TRANSLATION in den Variablenbereich der Firmware kopiert. Man kann den Platz danach also weiter verwerten.

Die Standardbelegung nach einem RESET ist:

alt	&A0	&A1	&A2	&A3	&A6	&AB	&AC	&AD	&AE	&AF
neu	&5E	&5C	&7B	&23	&40	&7C	&7D	&7E	&5D	&5B

Dadurch werden die in diesem Bereich liegenden Sonderzeichen solchen Zeichencodes zugeordnet, die nach ASCII-Norm in den verschiedenen Ländern unterschiedliche Zeichen erzeugen. In Deutschland sind das beispielsweise die Umlaute.

JUMPER (JUMP)
Eine Routine für die Sprungleiste

BD37 JUMP RESTORE Stelle die Original-Sprungleiste wieder her.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Mit Hilfe dieses Vektors kann die gesamte Sprungleiste im RAM ab &BB00 wieder in ihren Einschaltzustand versetzt werden. Alle bis dahin gemachten Patches werden wieder korrigiert. Die Sprungleiste ist bei allen CPCs verschieden lang:

- CPC 464: bis &BD3A (excl.)
- CPC 664: bis &BD5B (excl.)
- CPC 6128: bis &BD5E (excl.)

Die Indirections werden durch JUMP RESTORE nicht beeinflußt! Um auch diese wieder zu korrigieren, müssen die RESET- oder INTIALIZE-Vektoren der entsprechenden Packs aufgerufen werden.

Zusätzlich werden auch noch die BASIC-Vektoren zum Editor und den Rechenroutinen ins RAM kopiert. Diese Vektoren schliessen sich direkt an die Firmware-Sprungleiste an und gehen bis:

- CPC 464: bis &BDCD (excl.)
- CPC 664: bis &BDBE (excl.)
- CPC 664: bis &BDC1 (excl.)

Die Indirections der Firmware-Packs

BDCD IND TXT DRAW CURSOR Zeichne einen Cursor-Fleck, falls dieser auf beiden Ebenen eingeschaltet ist.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Wenn der Cursor sowohl "enabled" als auch "on" ist, wird die Cursorposition in das aktuelle Textfenster gezwungen und der Cursor gezeichnet.

BDD0 IND TXT UNDRAW CURSOR Entferne den Cursor-Fleck, falls dieser auf beiden Ebenen eingeschaltet ist.

Eingaben: keine
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Wenn der Cursor sowohl "enabled" als auch "on" ist, wird der Cursor-Fleck im aktuellen Textfenster wieder von der Cursorposition entfernt.

Bedingt durch den primitiven Aufbau des Standard-Cursors (Invertieren aller Bits der betroffenen Buchstabenfläche) wird für BDCD IND TXT DRAW CURSOR und &BDD IND TXT UNDRAW CURSOR dieselbe Routine benutzt!

BDD3 IND TXT WRITE CHAR Schreibe ein Zeichen auf den Bildschirm.

Eingaben: A = Zeichencode
H = Spalte (physikalisch)
L = Zeile (physikalisch)
Ausgaben: keine
Unverändert: IX, IY

Bezugspunkt für die Koordinatenangaben ist die linke obere Ecke mit den Koordinaten (0,0). Aufgabe dieser Indirection ist es, die Zeichenmatrix im ROM oder RAM zu suchen, entsprechend dem Bildschirmmodus zu expandieren, entsprechend PEN und PAPER einzufärben und entsprechend dem Vorder- und Hintergrund-Modus mit den alten Punkten auf der Zielposition zu verbinden.

Diese Indirection ist nicht zuständig für die Textausgabe auf der Grafikposition, Controlcodes, Kontrolle der Koordinaten auf Gültigkeit und den Cursor-Fleck (entfernen und wieder zeichnen).

BDD6 IND TXT UNWRITE Lies ein Zeichen vom Bildschirm.

Eingaben: H = Spalte (physikalisch)
L = Zeile (physikalisch)
Ausgaben: CY = 1 → A = Zeichencode
CY = 0 → kein Zeichen erkannt.
Unverändert: IX, IY

Bezugspunkt für die Koordinatenangaben ist die linke obere Ecke mit den Koordinaten (0,0). Der Cursor darf nicht auf der Leseposition dargestellt sein.

Um ein Zeichen vom Bildschirm zu lesen, wird die Grafikinformation auf der angegebenen Position wieder zu einer 8-Byte-Zeichenmatrix komprimiert und dann von 0 bis 255 mit allen Zeichen verglichen.

Dabei gibt es zwei Durchgänge: Zuerst wird die Zeichenposition unter der Annahme komprimiert, daß das Zeichen in der aktuellen PEN-Tinte geschrieben wurde. Wird so kein Zeichen erkannt oder ist das Ergebnis das Leerzeichen, so wird ein weiterer Durchgang unter der Annahme gestartet, daß der Hintergrund in der aktuellen PAPER-Tinte geschrieben ist.

Ziemlich viele Faktoren können diese Routine am Erfolg hindern:

Entweder wurden die Tinten gewechselt (oder auch nur einfach vertauscht), eine Zeichenmatrix geändert oder die Zeichenposition ganz oder teilweise durch Punkte oder Linien verändert. In all diesen Fällen erkennt diese Routine ein Zeichen nicht wieder.

BDD9 IND TXT OUT ACTION Schreibe ein Zeichen oder befolge einen Controlcode.

Eingaben: A = Zeichen- oder Controlcode
Ausgaben: keine
Unverändert: IX, IY

Diese Indirection übernimmt die gesamte Arbeit von &BB5A TXT OUTPUT. Der Vektor besorgt nur das Bankswitching und rettet die Register AF, BC, DE und HL.

BDDC IND GRA PLOT Zeichne einen Punkt.

Eingaben: DE = X-Koordinate
HL = Y-Koordinate
Ausgaben: keine
Unverändert: IX ,IY

Die Koordinatenangaben sind relativ zum Origin als Ursprung. Diese Indirection bewegt den Grafik-Cursor zur angegebenen Position und testet, ob die Koordinate im Grafikfenster liegt. Wenn ja, werden die Koordinaten in die entsprechende Byte-Adresse und Pixel-Maske umgerechnet, die aktuelle Gra-

fik-Vordergrund-Farbe wird expandiert und &BDE8 IND SCR WRITE aufgerufen. Diese Indirection befolgt noch den Grafik-Vordergrund-Modus (Force, Xor, And, Or) und setzt den Punkt.

BDDF IND GRA TEST Bestimme die Tinte eines Punktes.

Eingaben: DE = X-Koordinate
HL = Y-Koordinate
Ausgaben: A = Tintennummer des Punktes
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zum Origin als Nullpunkt. Diese Indirection bewegt den Grafik-Cursor zur angegebenen Position und testet, ob der Punkt im Grafik-Window liegt; wenn nicht, gibt sie die aktuelle PAPER-Tinte zurück. Wenn ja, werden die Koordinatenangaben in Byte-Adresse und Pixel-Maske umgerechnet, und &&BDE5 IND SCR READ wird aufgerufen, um die Tinte des Punktes zu bestimmen.

BDE2 IND GRA LINE Zeichne eine Linie.

Eingaben: DE = X-Koordinate
HL = Y-Koordinate
Ausgaben: keine
Unverändert: IX, IY

Die Koordinatenangaben sind relativ zum Origin. Der Grafik-Cursor wird auf die angegebene Position gesetzt und eine Linie von der alten Cursorposition zur neuen gezogen. Dabei werden nur Punkte innerhalb des aktuellen Grafikfensters gesetzt. Die einzelnen Punkte werden gesetzt, indem &BDE8 IND SCR WRITE aufgerufen wird, diese Routine wiederum befolgt den aktuellen Grafik-Vordergrund-Modus.

Beim CPC 664 und 6128 wird der erste Punkt der Linie nicht geplottet, wenn die Erster-Punkt-Option entsprechend gestellt ist. Außerdem werden nur die in der Linienmaske als Vordergrund-Pixel markierten Punkte mittels &BDE8 IND SCR WRITE gesetzt.

Hintergrund-Pixel werden entsprechend dem Grafik-Hintergrund-Modus entweder ignoriert (transparent) oder ohne Berücksichtigung der alten Tinte an dieser Position geplottet (opaque).

BDE5 IND SCR READ Lies einen Punkt vom Bildschirm.

Eingaben: HL = Byte-Adresse im Bildschirmspeicher
C = Pixel-Maske
Ausgaben: A = Tintennummer des Pixels
Unverändert: BC, DE, HL, IX, IY

BDE8 IND SCR WRITE Setze einen oder mehrere Punkte im Bildschirm unter Berücksichtigung des Grafikmodus.

Eingaben: HL = Byte-Adresse im Bildschirmspeicher
C = Maske für das oder die Pixel
B = expandierte Tinte (Farb-Byte)
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Das oder die durch HL adressierten und C maskierten Pixel werden entsprechend dem aktuellen Grafik-Vordergrund-Modus mit dem alten Bildschirminhalt verknüpft und in den Bildschirm geplottet.

BDEB IND SCR MODE CLEAR Lösche den Bildschirm mit Tinte 0.

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Diese Indirection führt alle Aktionen für &BC14 SCR CLEAR durch.

BDEE IND KM TEST BREAK Führe den Test auf BREAK und RESET durch.

Eingaben: C = Status von SHIFT und CTRL
Ausgaben: keine
Unverändert: BC, DE, IX, IY

Diese Routine wird vom Interrupt-Pfad aus aufgerufen (&BDF4 IND KM SCAN KEYS beim CPC 664 und 6128). Deshalb müssen Interrupts verboten sein und bleiben es auch.

Der SHIFT- und CTRL-Status ist in C folgendermaßen codiert:

Bit 5 = 1 – SHIFT ist gedrückt
Bit 7 = 1 – ESCape ist gedrückt

Der Zustand der ESC-Taste wird noch einmal überprüft. Sind nur ESC, CTRL und SHIFT gedrückt, wird ein Restart 0 durchgeführt (Kaltstart).

Ist die ESC-Taste gedrückt, die weiteren Bedingungen aber nicht erfüllt und der Break-Mechanismus aktiv, so werden die Aktionen wie bei &BB4B KM BREAK EVENT beschrieben durchgeführt.

BDF1 IND MC WAIT PRINTER Versuche, ein Zeichen zum Drucker zu schicken.

Eingaben: A = Code des zu druckenden Zeichens
Ausgaben: CY = 1 → Zeichen wurde zum Drucker gesandt
CY = 0 → Es dauerte zu lange, nicht gedruckt
Unverändert: DE, HL, IX, IY

Diese Indirection führt alle Aktionen für &BD2B MC PRINT CHAR aus. Dieser Vektor blendet nur das untere ROM ein und rettet auch noch das BC-Register.

Beim CPC 464 reicht der ungenutzte Platz bis &BDFF genau aus, um eine 2-Pfennig-Lösung für das 8. Drucker-Bit Software-mäßig zu unterstützen. Beim CPC 664 und 6128 ist jetzt leider die folgende Indirection im Weg.

BDF4 IND KM SCAN KEYS Frage die Tastatur ab (nur CPC 664 und 6128).

Eingaben: keine
Ausgaben: keine
Unverändert: IX, IY

Bei Aufruf dieser Indirection muß der Interrupt verboten sein und bleibt es auch.

Die Tastaturmatrix wird überprüft, und neu gedrückte Tasten werden in den Tastaturpuffer eingetragen.

Ist ESC gedrückt, wird &BDEE IND TEST BREAK aufgerufen.

Diese Routine kümmert sich auch um die Entprellung der Tastatur (indem losgelassene Tasten erst nach zwei Durchläufen als nicht mehr gedrückt markiert werden) und um das Auto-Repeat der einzelnen Tasten.

THE HIGH KERNEL JUMPBLOCK
Die obere Sprungleiste des Kernel

B900 HI KL U ROM ENABLE Blende das momentan selektierte obere ROM ein.

Eingaben: keine
Ausgaben: A = alter ROM-Status
Unverändert: BC, DE, HL, IX, IY

Das momentan im obersten Adreßviertel angewählte ROM wird eingeblendet. Lesezugriffe der CPU über &C000 lesen nun die Werte aus diesem ROM. Der Ausgabewert in A kann benutzt werden, um später mit &B90C HI KL ROM RESTORE den alten ROM-Status wieder herzustellen.

B903 HI KL U ROM DISABLE Blende das obere ROM aus und RAM ein.

Eingaben: keine
Ausgaben: A = alter ROM-Status
Unverändert: BC, DE, HL, IX, IY

Das momentan im obersten Adreßblock angewählte ROM wird ausgeblendet. Lesezugriffe der CPU über &C000 beziehen sich nun auf das RAM (normalerweise der Bildwiederholspeicher). Der Ausgabewert in A kann benutzt werden, um zu einem späteren Zeitpunkt mit &B90C HI KL ROM RESTORE den alten ROM-Status wiederherzustellen.

B906 HI KL L ROM ENABLE Blende unten das Betriebssystems-ROM ein.

Eingaben: keine
Ausgaben: A = alter ROM-Status
Unverändert: BC, DE, HL, IX, IY

Das Betriebssystem-ROM wird eingeblendet. Lesezugriffe der CPU unterhalb von &4000 erhalten nun ihre Werte aus diesem ROM. Normalerweise wird

dieses ROM nur eingeblendet, wenn eine Betriebssystem-Routine via Restart aufgerufen wird, ansonsten findet man unten nur RAM. Der Ausgabewert im A-Register kann hinterher an &B90C HI KL ROM RESTORE übergeben werden, um den alten ROM-Status wieder herzustellen.

B909 HI KL L ROM DISABLE Blende das untere ROM wieder aus und RAM ein.

Eingaben: keine
Ausgaben: A = alter ROM-Status
Unverändert: BC, DE, HL, IX, IY

Dieser Vektor blendet das untere ROM (Betriebssystem) wieder aus. Der Ausgabewert im A-Register kann ebenfalls wieder benutzt werden, um mit &B90C HI KL ROM RESTORE den alten ROM-Status wiederherzustellen.

B90C HI KL ROM RESTORE Stelle eine frühere ROM-Konfiguration wieder her.

Eingaben: A = alter ROM-Status bzw. ROM-Selection
Ausgaben: keine
Unverändert: BC, DE, HL, IX, IY

Mit Hilfe dieses Vektors kann eine alte ROM-Konfiguration, die im A-Register übergeben wird, wiederhergestellt werden. Da ein Programm, das beispielsweise das untere ROM mit &B906 HI KL L ROM ENABLE eingeblendet hat, nicht sicher wissen kann, daß dieses ROM auch vorher wirklich nicht eingeblendet war, ist es mitunter höchst unsicher, das ROM nachher einfach mit 6B909 HI KL L ROM DISABLE wieder auszublenden.

B90F HI KL ROM SELECT Wähle ein bestimmtes ROM an, und blende es ein.

Eingaben: C = ROM-Select-Byte
Ausgaben: C = alte ROM-Selection
B = alter ROM-Status
Unverändert: DE, HL, IX, IY

Entsprechend C wird im obersten Speicherblock ein neues ROM angewählt und eingeblendet. Die Ausgabewerte reflektieren die alte ROM-Konfigurati-

on, die sich aus ROM-Selektion und ROM-Status zusammensetzt. Diese Werte kann man an &B918 HI KL ROM DESELECT übergeben, um später die alte ROM-Konfiguration wiederherzustellen.

B912 HI KL CURR SELECTION Frage an, welches ROM oben gerade selektiert ist.

Eingaben: keine
Ausgaben: A = aktuelles ROM-Select-Byte
Unverändert: F, BC, DE, HL, IX, IY

Es ist nicht nur für andere Programme interessant, welches ROM im oberen Adreßblock gerade angewählt (aber nicht unbedingt eingeblendet) ist, auch dem Programm in diesem ROM selbst kann das mitunter noch unbekannt sein.

B915 HI KL PROBE ROM Bestimme Klasse & Version eines ROMs.

Eingaben: C = ROM-Select-Byte
Ausgaben: A = ROM Class
L = Mark Number
H = Version Number
Unverändert: C, DE, IX, IY

Dieser Vektor blendet das gewünschte ROM kurzzeitig ein, um die ersten drei Bytes daraus zur Verfügung zu stellen. Diese enthalten einige mehr oder weniger wichtige statistische Informationen über das ROM. Interessant ist nur die ROM CLASS, die immer auf der ersten Adresse des ROMs (&C000) zu finden ist:

A = 0 – Vordergrundprogramm
A = 1 – Hintergrundprogramm
A = 2 – Erweiterungs-ROM (Zusatz-ROM zu 0 oder 1)
A = &80 – Das gesetzte 7. Bit kennzeichnet das eingebaute ROM des BASIC-Interpreters. Fühlt sich von der ausgegebenen ROM-Adresse kein extern angeschlossenes ROM angesprochen, wird automatisch dieses ROM eingeblendet. Der BASIC-Interpreter ist also auf jeder unbenutzten ROM-Adresse ansprechbar.

B918 HI KL ROM DESELECTION Stelle eine frühere ROM-Selektion wieder her.

Eingaben: C = alte ROM-Selektion
B = alter ROM-Status
Ausgaben: C = zuletzt angewähltes ROM
Unverändert: AF, DE, HL, IX, IY

Mit diesem Vektor wird die durch &B90F HI KL ROM SELECT vorgenommene Veränderung der ROM-Konfiguration wieder rückgängig gemacht.

B91B HI KL LDIR Führe ein LDIR im RAM durch.

Eingaben: HL = Adresse des ersten Quellbytes
DE = Adresse des ersten Zielbytes
BC = Länge des zu verschiebenden RAM-Bereichs
Ausgaben: wie nach LDIR
Unverändert: wie nach LDIR

B91E HI KL LDDR Führe ein LDDR im RAM durch.

Eingaben: HL = Adresse des ersten Quellbytes
DE = Adresse des ersten Zielbytes
BC = Länge des zu verschiebenden RAM-Bereiches
Ausgaben: wie nach LDDR
Unverändert: wie nach LDDR

B921 HI KL POLL SYNCHRONOUS Teste, ob eine synchronisierbare Unterbrechung auf ihre Ausführung wartet.

Eingaben: keine
Ausgaben: CY=0 → Synchronous Event Pending Queue ist leer
CY=1 → In der Queue wartet ein Event, das behandelt werden sollte
Unverändert: BC, DE, HL, IX, IY

Diese RAM-residente Routine dient dazu, mit möglichst wenig Zeitverlust die Synchronous Event Pending Queue zu pollen. Dies kann auch aus einer Event-Behandlungsroutine heraus geschehen. In diesem Fall versteckt der Kernel alle Events mit niedrigerer Priorität als die des laufenden Events. Wartet ein Event (mit höherer Priorität) auf seine Ausführung, so sollten danach &BCFB KL NEXT SYNC, &BCFE KL DO SYNC und &BD01 KL DONE SYNC aufgerufen werden.

B92A HI KL SCAN NEEDED Teile dem Kernel mit, daß die Tastatur mit dem nächsten Interrupt abgefragt werden muß (nur CPC 664 und 6128).

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, IX, IY

Diese Routine setzt den Systemspeicher des Frequenzteilers für den Ticker-Interrupt auf Eins. Dadurch wird der Kernel glauben gemacht, die Ticker Chain sei bereits beim nächsten Interrupt wieder dran. Da die Tastaturabfrage völlig systemkonform als Ticker Event eingebunden ist, wird auch die Tastatur beim nächsten Interrupt abgefragt.

Wiederholtes Aufrufen dieses Vektors kann die Geschwindigkeit der Ticker-Liste bis auf 300 Kicks pro Sekunde hochtreiben.

THE LOW KERNEL JUMPBLOCK
Die untere Sprungleiste des Kernel

0000 LOW RESET ENTRY
RST 0 Kaltstart, totaler Reset.

Eingaben: keine
Ausgaben: Routine kehrt nicht zurück
Unverändert: –/–

Die Hardware wird (fast) vollständig zurückgesetzt, die Firmware komplett initialisiert und dann das Vordergrund-Programm in ROM 0 (BASIC) gestartet. Im CPC 6128 wird zusätzlich die normale RAM-Bank angewählt.

Der FDC kann nicht 100%ig zurückgesetzt werden, da sein RESET-Eingang nur mit dem Einschalt-RESET verbunden ist. Hat man ihn beispielsweise auf DMA-Modus eingestellt, hilft nur noch Aus- und Wiedereinschalten des Computers:

```
OUT &FB7F,3          ' Laufwerkparameter setzen
OUT &FB7F,0
OUT &FB7F,0          ' u. a. DMA-Betrieb
CAT                  ' Aerger
CTRL/SHIFT/ESCAPE    ' wieder Ruhe
CAT                  ' nach wie vor: Aerger
```

Wie man sieht, hält sich AMSTRAD auch nicht an die selbst erfundenen Regeln.

0008 LOW LOW JUMP
RST 1

Sprung zu einer Routine im unteren ROM oder RAM mit Angabe des gewünschten ROM-Status und Routinenadresse in einem nachgestellten DEFW xxxx.

Eingaben: 2-Byte-Inline-Adresse
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Anwendung:
```
RST  1                          ; erweiterter JP-Befehl
DEFW ADRESSE+&8000+&4000
```

Dem Restart wird ein 2-Byte-Vektor mit einer Adresse im unteren ROM oder RAM angehängt. Bits 14 und 15 der Adresse sind dabei immer null. Diese Bits werden vom Restart 1 als ROM-Status interpretiert:

Bit 14 = 0 – unten (bis &3FFF) das ROM einblenden
Bit 14 = 1 – unten das ROM ausblenden
Bit 15 = 0 – oben (ab &C000) das selektierte ROM einblenden
Bit 15 = 1 – oben das ROM ausblenden

Dieser Vektor simuliert den Z80-Befehl JP ADRESSE, nur daß er auch noch den ROM-Status wie gewünscht einstellt. Es werden keine Register verändert.

Ausgenommen davon ist der zweite Registersatz: Alle Restarts schalten mit EXX die beiden Registersätze um, um notwendige Operationen durchführen zu können, ohne die normalen Register zu verändern. Deshalb muß auch immer der Interrupt verboten und nachher wieder zugelassen werden. Alle Restarts schalten deshalb den Hardware-Interrupt wieder ein!

Kehrt die so angesprungene Routine mit RET wieder zurück, so läuft sie in eine manipulierte Rücksprungadresse, die zuerst den alten ROM-Status wiederherstellt.

Auch hierbei werden Interrupts wieder zugelassen, andererseits aber keine Register verändert. Die zurückgegebenen Register stammen alle von der aufgerufenen Routine.

Dieser Restart ist für den Einsatz in Sprungleisten im RAM gedacht und wird im Firmware Jumpblock exzessiv genutzt.

000B LOW KL LOW PCHL Sprung zu einer Routine im unteren ROM oder RAM mit Angabe der gewünschten ROM-Konfiguration und Adresse im HL-Register.

Eingaben: HL = ROM-Status und Routinenadresse
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Dieser Vektor entspricht fast völlig dem RST_1 &08 LOW LOW JUMP. Die Routinenadresse muß nur im HL-Register übergeben werden. Es können also nur Routinen angesprungen werden, die im HL-Register keine Eingabeparameter erwarten.

000E LOW PCBC INSTRUCTION JP (BC)

Eingaben: BC = Routinenadresse
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

An dieser Stelle stehen die beiden Z80-Befehle:

```
000E PUSH BC
000F RET
```

wodurch der Sprung zur angegebenen Adresse erreicht wird.

0010 LOW SIDE CALL RST 2 Aufruf einer Routine in einem benachbarten oberen ROM mit Angabe der "Distanz" und Adresse in einem nachgestellten DEFW xxxx.

Eingaben: 2-Byte-Inline-Adresse
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX

Anwendung:
```
RST  2                        ; erweiterter CALL-Befehl
DEFW OFFSET*&4000 + Adresse-&C000
```

Für Modul-Programme, die den Rahmen eines 16k-Eproms sprengen, unterstützt der Kernel bis zu 3 Erweiterungs-ROMs pro Vordergrund-ROM. Programme können so bis zu 64 kByte ROM umfassen.

Beim Ein- und Aussprung werden wieder alle Register bis auf IY von und zur gerufenen Routine weitergegeben. Das gewünschte ROM wird angewählt (selektiert) und eingeblendet und das untere ROM mit dem Betriebssystem ausgeblendet. Der Stack wird dabei so manipuliert, daß die aufgerufene Routine, sobald sie mit RET abschließt, zuerst noch in eine andere Routine läuft, die die alte ROM-Konfiguration wiederherstellt.

Der Restart 2 simuliert einen erweiterten CALL-Befehl. Das angehängte Word enthält dabei die gewünschte Routinenadresse. Da der SIDE CALL nur für den oberen Speicherblock (ab &C000) zulässig ist, wären die Bits 14 und 15 immer gesetzt. In diesen Bits kann deshalb ein ROM-Nummern-Offset angegeben werden.

Bezugspunkt ist dabei die ROM-Nummer des laufenden Vordergrund-Programms. Umfaßt ein Programmpaket beispielsweise 4 ROMs (4*16k = 64k), so darf nur das erste als Vordergrund-ROM markiert werden (ROM-Typ im ersten Byte = Adresse &C000). Die anderen drei ROMs müssen die nächsten folgenden ROM-Select-Adressen belegen und erhalten eine Kennung als Erweiterungs-ROM.

Wird das Vordergrund-Programm gestartet, merkt sich der Kernel die ROM-Adresse, die dem Programm selbst vollkommen unbekannt bleiben kann (unbekannter Modulschacht). Soll eine Routine im 2. Erweiterungs-ROM aufgerufen werden, geht das einfach via SIDE CALL: Die obersten beiden Bits erhalten den ROM-Offset 2 (&X10). Diese Routine selbst kann nun ein Unterprogramm im ersten Erweiterungs-ROM aufrufen: ROM-Offset ist jetzt 1 (&X01) usw.

0013 LOW KL SIDE PCHL Aufruf einer Routine in einem benachbarten oberen ROM mit Angabe der "Distanz" und Adresse im HL-Register.

Eingaben: HL = Routinenadresse + ROM-Offset
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX

Dieser Vektor entspricht fast völlig dem vorhergehenden RST 2 LOW SIDE CALL.

Der einzige Unterschied liegt in der Übergabe der Routinenadresse, die hier im HL-Register erfolgt. Über diesen Vektor können deshalb keine Routinen aufgerufen werden, die im HL-Register Parameter erwarten.

0016 LOW PCDE INSTRUCTION JP (DE)

Eingaben: DE = Routinenadresse
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Diese Routine springt zur im DE-Register angegebenen Adresse. Diese wird durch die folgenden beiden Befehle erreicht:

```
0016 PUSH DE
0017 RET
```

0018 LOW FAR CALL RST 3

Aufruf einer Routine im RAM oder jedem beliebigen ROM. ROM-Selektion und Adresse werden indirekt über ein nachgestelltes DEFW xxxx angezeigt.

Eingaben: Ein Inline-angegebener Vektor zeigt auf die 3-Byte-Far Address
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX

Anwendung:

```
        RST  3              ; erweiterter CALL-Befehl
        DEFW FARADR         ; Zeiger auf die Far Address
        ...

FARADR: DEFW ADRESSE        ; Routinenadresse
        DEFB ROMKONFIG      ; benötigte ROM-Konfiguration
```

Dieser Restart ist das Arbeitspferd unter den "Befehlserweiterungen" für den Z80. Hiermit läßt sich jede Routine in jedem ROM aufrufen.

An den Restart muß dabei direkt ein Zeiger auf die sogenannte Far Address angehängt werden. Hier steht dann die Routinenadresse und ein Byte, das die ROM-Konfiguration bestimmt.

Auch bei diesem Restart werden alle Register unberührt hin- und zurückgegeben. Ausgenommen davon ist nur wieder das IY-Register: Wird eine Routine in einem Hintergrund-ROM aufgerufen, so wird auf dem Hinweg das IY auf die Untergrenze seines oberen RAM-Bereichs gesetzt. Das ist der Wert plus 1, der nach der Initialisierung des ROMs mit &BCCE KL INIT BACK im HL-Register zurückgegeben wird.

Auch hier wird wieder der Stack so manipuliert, daß nach dem RET der aufgerufenen Routine zuerst noch die alte ROM-Konfiguration wiederhergestellt wird.

Das Konfigurationsbyte in der Far Address kann dabei auf zwei unterschiedliche Weisen interpretiert werden: Für Werte von 0 bis 251 wird das ROM mit dieser Nummer selektiert, eingeblendet und unten das Betriebssystem-ROM ausgeblendet. Existiert kein externes ROM an der angegebenen Position, so wird automatisch das eingebaute BASIC-ROM angewählt.

Bei Werten größer als 251 wird kein neues ROM angewählt, sondern nur der ROM-Status geändert:

252 – unten ROM – oben ROM
253 – unten RAM – oben ROM
254 – unten ROM – oben RAM
255 – unten RAM – oben RAM

001B LOW KL FAR PCHL Aufruf einer Routine in RAM oder jedem beliebigen ROM. Die Register C und HL enthalten die gewünschte ROM-Selektion und Adresse.

Eingaben: HL = Routinenadresse
C = ROM-Konfiguration
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX

Dieser Vektor entspricht wieder dem vorhergehenden RST 3 FAR CALL. Einziger Unterschied liegt wieder in der Übergabe der Routinenadresse, wofür diesmal die Register HL und C herhalten müssen. Mit diesem Vektor können also keine Routinen aufgerufen werden, die in HL oder C Eingabeparameter erwarten.

001E LOW PCHL INSTRUCTION JP (HL)

Eingaben: HL = Routinenadresse
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

An dieser Stelle steht lediglich der Z80-Befehl JP (HL). Dieser Vektor ist wie &0E LOW PCBC INSTRUCTION und &16 LOW PCDE INSTRUCTION

sinnvoll, weil man durch Aufruf dieses Vektors (CALL #001E) den nicht existenten Z80-Befehl CALL (HL) simulieren kann.

0020 LOW RAM LAM LD A,(HL) aus dem RAM.
RST 4

Eingaben: HL = RAM-Adresse
Ausgaben: A = Inhalt dieser Speicherzelle
Unverändert: F, BC, DE, HL, IX, IY

Dieser Restart gibt ROM-Programmen eine bequeme Möglichkeit, unabhängig vom aktuellen ROM-Status aus dem RAM zu lesen. Dieser Restart ersetzt dabei den Z80-Befehl LD A,(HL).

Ein entsprechender Vektor zum Beschreiben des RAMs ist nicht notwendig, da die jeweils eingestellte ROM-Konfiguration nur von den Lesebefehlen der CPU beeinflußt wird. Schreibbefehle gehen immer an das eingebaute RAM.

0023 LOW KL FAR ICALL Aufruf einer Routine im RAM oder jedem beliebigen ROM. ROM-Selektion und Adresse werden vom HL-Register angezeigt.

Eingaben: HL zeigt auf die 3-Byte-Far Adress
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX

Auch dieser Vektor entspricht dem Restart 3 FAR CALL. Der Unterschied ist wieder die Art, wie die Routinenadresse übergeben wird. Diesmal wird das HL-Register benutzt, um auf die Far Address zu zeigen. Das HL-Register kann also wieder nicht benutzt werden, um Parameter zur aufgerufenen Routine zu übergeben.

0028 LOW FIRM JUMP Sprung zu einer Routine im unteren
RST 5 ROM. Die gewünschte Adresse wird in einem DEFW xxxx angehängt.

Eingaben: Inline-angegebener Vektor zur Routine
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Anwendung:
```
RST  5                    ; erweiterterJP-Befehl
DEFW ADRESSE
```

Die angegebene Routine wird aufgerufen, wobei wieder alle Register von und zur Routine unverändert weitergereicht werden.

Bevor die Routine angesprungen wird, wird das untere ROM eingeblendet (die Routine selbst kann dabei durchaus über &4000 liegen!). Nachher wird das untere ROM *immer* ausgeblendet, unabhängig vom alten ROM-Status. Normalerweise ist das unkritisch, da beim "Standard-CPC-Betrieb" das Betriebssystem-ROM immer ausgeschaltet sein soll.

Schreibt man aber ein Programm so, daß das untere ROM die meiste Zeit eingeblendet ist, muß man beim Aufruf einiger Vektoren, die diesen Restart benutzen, Vorsicht walten lassen.

0030 LOW USER RESTART
RST 6

Vom Betriebssystem nicht benutzt. Der RST 6 vom ROM speichert jedoch den momentanen RAM- und ROM-Status in Adresse &28, blendet RAM ein und springt den RST 6 im RAM an.

Eingaben: ?
Ausgaben: ?
Unverändert: ?

Dieser Restart kann von einem Vordergrund-Programm mit einer eigenen Aufgabe versehen werden.

Da man Änderungen am Maschinencode an dieser Stelle nur im RAM vornehmen kann, wurde ein spezieller Mechanismus implementiert: Der RST 6 im ROM speichert den aktuellen ROM-Status in der Speicherstelle &2B, blendet das untere ROM aus und wiederholt den Restart. Dadurch kann man einerseits unabhängig vom ROM-Status auf den Restart 6 zugreifen und andererseits nach Beendigung dieser Routine den alten ROM-Status wiederherstellen. Denkbar ist beispielsweise der Einsatz dieses Restarts als Break-point in einem Maschinensprache-Monitor.

0038 LOW INTERRUPT ENTRY

Der Z80 wird im Interrupt-Modus 1 RST 7 betrieben. Das heißt, daß er mit jeder Interrupt-Anforderung einen RST 7 ausführt.

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

Jeder Hardware-Interrupt führt zu einem Unterprogramm-Aufruf an dieser Stelle.

Die Behandlungsroutine muß zunächst zwischen einem normalen Timer-Interrupt der ULA und einer möglichen Interrupt-Anforderung eines externen Gerätes unterscheiden. Dazu wird in der Interrupt-Routine sofort ein Interrupt kurzzeitig wieder zugelassen. Liegt das Interrupt-Signal immer noch an, so wird der Interrupt wieder unterbrochen, woran letztendlich ein externer Interrupt erkannt wird.

Wenn nicht, war es ein normaler Timer-Interrupt. Dann wird ein Zähler erhöht und die eine oder andere Chain für die Software-Interrupts abgearbeitet. Vom Kernel selbst werden hier folgende Blöcke eingehängt:

- Tastatur-Abfrage (Ticker)
- Farbblinken (Frame Flyback)

003B LOW EXT INTERRUPT Erkennt der Kernel ein Interrupt-Signal von einer Erweiterung am Systembus, so wird diese Adresse angesprungen.

Eingaben: keine (A' = Wert aus C')
Ausgaben: keine
Unverändert: IX, IY

Erkennt die normale Interrupt-Routine eine externe Interrupt-Anforderung, so wird das untere ROM ausgeblendet und dieser Vektor angesprungen.

Die hier installierte Routine sollte:

1. nicht gepatcht werden, ohne vorher eine Kopie des alten Eintrags an dieser Stelle (5 Bytes) gemacht zu haben. Dieser Vektor muß deshalb ortsunabhängig sein.

2. nachschauen, ob der Interrupt auch wirklich für sie ist. Wenn nicht, dann muß die Kopie des Vektors aufgerufen werden.

3. den Interrupt löschen.

4. keinesfalls einen Interrupt wieder zulassen oder ein Register dauerhaft ändern, außer: AF, BC, DE und HL.
5. möglichst schnell abschließen, damit möglichst keine Timer-Interrupts ausgelassen werden.

Nach Möglichkeit sollte in der Interrupt-Routine selbst nur ein Event gekickt werden: &BCF2 KL EVENT. Dazu muß aber das untere ROM wieder eingeblendet werden, wozu man den Wert aus C' (aktueller ROM-Status und Bildschirm-Modus) benötigt. Dieser Wert ist hier jedoch in A' gerettet.

Die BASIC-Vektoren

Die BASIC-Vektoren haben, im Gegensatz zu allen anderen, beim CPC 464, 664 und 6128 nicht die gleiche Lage. Zu jedem Vektor sind deshalb alle drei Einsprungsstellen angegeben in der Reihenfolge 464–664–6128.

Außerdem wurden beim CPC 664 und 6128 die Integervektoren weggelassen, weil die entsprechenden Routinen in das BASIC-ROM verlegt wurden. Hier ist dann jeweils die Routinenadresse angegeben. Bevor man diese Routinen aber aufrufen kann, muß man auch das BASIC-ROM einblenden! Allgemein kann man sich hier eines Restarts bedienen.

Einige Fließkomma-Vektoren wurden ebenfalls nicht mehr in den BASIC-Jumpblock aufgenommen.

DER EDITOR

BD3A BD5B BD5E EDIT Editieren bzw. Neu-Eingabe eines Strings (Zeichenkette).

Eingaben: HL zeigt auf den Puffer
Ausgaben: CY = 1 → o.k. (Anwender schloß mit ENTER ab)
CY = 0 → BREAK (Anwender schloß mit ESC ab)
Unverändert: BC, DE, HL, IX, IY

Mit Hilfe des Editors kann ein beliebiger Text verändert oder neu eingegeben werden. Der Text wird dem Editor in einem Puffer bereitgestellt und kann bis zu 255 Zeichen lang sein. Abgeschlossen werden muß er mit einem Null-Byte. Der Puffer muß immer 256 Bytes lang sein und darf durch kein ROM verdeckt werden (er kann deshalb nie unterhalb von &4000 liegen). Soll ein Text nicht verändert, sondern ein neuer eingegeben werden, so muß man praktisch einen null Zeichen langen Text "editieren": Das erste Byte des Puffers, auf das HL zeigt, muß auf Null gesetzt sein.

Sobald der Editor aufgerufen wurde, gibt dieser selbst den Text ab der aktuellen Cursorposition im aktuellen Textfenster aus. Dabei benutzt er &BB5D TXT WR CHAR. Sonderzeichen werden deshalb nie befolgt, sondern immer als Grafikzeichen ausgedruckt. Ausgenommen davon sind die Codes 0, 13 und 16, die der Editor selbst als Steuerzeichen interpretiert. Abgesehen von CHR$ (0) können aber alle Zeichen mit Hilfe des Copy-Cursors eingegeben werden.

Der Editor beschränkt sich auf das aktuelle Textfenster. Ist dies zu klein, den ganzen Text zu fassen, scrollt er auch selbsttätig darin. Leider kann der Copy-Cursor auch nur innerhalb dieses Textfensters bewegt werden.

Der Editor des Schneider CPC 664 und 6128 wurde um eine nützliche Funktion erweitert: Beim CPC 464 fällt es meist unangenehm auf, daß nach der Eingabe von "EDIT zeile" der Cursor nicht auf dem ersten Zeichen der BASIC-Zeile, wie von AUTO her gewohnt, sondern auf dem ersten Zeichen der Zeilennummer steht. Bei CPC 664 bzw. 6128 testet der Editor jetzt erst, ob der zu bearbeitende Text mit einer Zahl anfängt. Diese interpretiert er dann als Zeilennummer und stellt den Cursor bereits hinter die Zahl.

Außerdem enthält der Editor beim CPC 464 und 664 einen kleinen Fehler. Normalerweise kann der Anwender den Editor mit CTRL plus TAB zwischen Einfüge- und Überschreibmodus hin- und herschalten. Ist der bearbeitete String aber gerade leer, so funktioniert es beim CPC 464 und 664 nicht korrekt: Scheinbar ist der Editor der Meinung, man habe ENTER (zu Insert zurückschalten) bzw. ESC (Insert abschalten) gedrückt und terminiert entsprechend. Dieser Fehler wurde von AMSTRAD erst beim CPC 6128 korrigiert.

DIE FLIESSKOMMA-ROUTINEN

Das Fließkomma-Pack im unteren ROM stellt alle gebräuchlichen Rechenfunktionen und -operationen zur Verfügung. Dabei gelten folgende Ein- und Ausgabeschnittstellen:

Bevorzugter Eingabeparameter ist FLO(HL). Bei den Operationen kommt noch FLO(DE) als zweiter Parameter dazu. Die Ausgabe erfolgt immer in FLO(HL), wenn das ausreicht, aber auch manchmal direkt im A-Register. Der Wert des HL-Registers ändert sich nicht, d.h. die Ausgabe des Ergebnisses erfolgt in dem Fließkommaspeicher, der das bzw. eins der beiden Argumente enthielt.

Rechenroutinen, die versagen können, liefern im Carry-Flag den Fehlerstatus: Wie fast überall im Betriebssystem bedeutet CY=1, daß kein Fehler vorliegt,

und CY=0, daß ein Fehler auftrat. In letzterem Fall können die folgenden Flags noch zusätzliche Informationen enthalten:

Z = 1 – Division durch Null
S = 1 – ungültiger Parameter
P = 1 – Überlauf

Die Fließkommaspeicher dürfen nicht durch ein ROM verdeckt sein. Normalerweise dürfen also unterhalb von &4000 keine Speicher liegen. Ausnahme davon sind Konstanten, die im unteren ROM liegen und auf die die FLO-Routinen zugreifen können.

Zufallsgenerator

Diese vier Routinen stellen einen komfortablen Zufallszahlengenerator dar. Als Grundlage dient eine "Seed" genannte 4-Byte breite Zahl, ein sogenanntes "Long Word". Die Zufallszahlen, die erzeugt werden, sind aber vom Typ "Real", also Fließkomma, und liegen im Bereich 0 <= RND < 1.

Da es einem digitalen Automaten vom Prinzip her sehr schwerfällt, wirklich zufällige Zahlen bzw. Zahlenfolgen zu erzeugen, benutzt man auch im Schneider CPC dafür eine recht gebräuchliche Rechenvorschrift. Das hat zur Folge, daß man nach einem RANDOMIZE mit einem konstanten Initialisierungswert mit RND immer dieselben Zahlenfolgen erhält. Dieser Effekt ist letztendlich aber auch sehr nützlich, wenn man in einem "zufallsgesteuerten" Programm einem Fehler auf die Spur kommen will.

BD97 BDB8 BDBB FLO RANDOMIZE 0 &89656C07 → LW(seed)

Eingaben: keine
Ausgaben: keine
Unverändert: AF, BC, DE, IX, IY

BD9A BDBB BDBE FLO RANDOMIZE
FLO(HL) XOR &89656C07 → LW(seed)

Eingaben: FLO(HL) = Fließkommazahl
Ausgaben: keine
Unverändert: C, IY, FLO(HL)

BD9D BD7C BD7F FLO RND RND → FLO(HL)

Eingaben: FLO(HL) = Fließkommaspeicher für die Zufallszahl
Ausgaben: FLO(HL) = nächste Zufallszahl
Unverändert: HL, IY

BDA0 BD88 BD8B FLO LAST RND letzten RND-Wert → FLO(HL)

Eingaben: FLO(HL) = Fließkommaspeicher für die Zufallszahl
Ausgaben: FLO(HL) = alte Zufallszahl
Unverändert: HL, IY

Operationen

Es folgen die Operationen, die vom Fließkomma-Pack bereitgestellt werden.

Beim CPC 664 und 6128 wurde der Vektor für FLO SUB ersatzlos gestrichen, aber FLO SUB* belassen. Das ist insofern unglücklich, als FLO SUB den IN/OUT-Standard exakt befolgt hätte, nicht aber FLO SUB*, wo FLO(HL) nicht mehr das "erste" Argument ist.

FLO POT enthält zwei kleine Fehler: 0 hoch 0 wird kommentarlos zu 1 evaluiert. Das ist falsch, denn 0° ist nicht definiert.

Bei einem Überlauf wird normalerweise als "Ergebnis" immer die größte, darstellbare Zahl vorzeichenrichtig ausgegeben. Bei einer negativen Basis und einem ganzzahligen geraden Exponenten müßte deshalb +1.70141E+38 ausgegeben werden. Tatsächlich wird aber auch hier der negative Wert angezeigt:

```
PRINT -10 ^ +40 → Overflow  -1.70141E+38
```

Das resultiert wahrscheinlich aus einem anderen Service, den diese Routine bereitstellt: Wird zu einer negativen Basis ein nicht ganzzahliger Exponent angegeben, quittiert FLO POT mit M (Signum: S=1), also ungültiger Parameter. Die Routine bricht die Berechnung aber nicht kommentarlos ab, sondern berechnet einfach den Funktionswert zur entsprechenden positiven Basis und gibt dem Funktionswert ein negatives Vorzeichen. BASIC schert sich in diesem Fall nicht um die Fehlermeldung und übernimmt kommentarlos diesen Wert:

```
PRINT -10 ^ 3.3  → -1995.26231
```

BD58 BD79 BD7C FLO ADD FLO(HL) + FLO(DE) → FLO(HL)

Eingaben: FLO(HL) und FLO(DE)
Ausgaben: FLO(HL)
CY = 0 → Überlauf
Unverändert: HL, FLO(DE)

BD5E BD7F BD82 FLO SUB• FLO(DE) – FLO(HL) → FLO(HL)

Eingaben: FLO(HL) und FLO(DE)
Ausgaben: FLO(HL)
CY= 0 → Überlauf
Unverändert: HL, FLO(DE)

BD5B 349A 349A FLO SUB FLO(HL) – FLO(DE) → FLO(HL)

Eingaben: FLO(HL) und FLO(DE)
Ausgaben: FLO(HL)
CY = 0 → Überlauf
Unverändert: HL, FLO(DE)

BD61 BD82 BD85 FLO MULT FLO(HL) * FLO(DE) → FLO(HL)

Eingaben: FLO(HL) und FLO(DE)
Ausgaben: FLO(HL)
CY = 0 → Überlauf
Unverändert: HL, FLO(DE)

BD64 BD85 BD88 FLO DIV FLO(HL) / FLO(DE) → FLO(HL)

Eingaben: FLO(HL) und FLO(DE)
Ausgaben: FLO(HL)
CY = 0 → Überlauf, wenn auch Z=1, dann Division durch Null
Unverändert: HL, FLO(DE)

BD7C BD9D BDA0 FLO POT FLO(HL) ^ FLO(DE) → FLO(HL)

Eingaben: FLO(HL) und FLO(DE)
Ausgaben: FLO(HL)
CY = 1 → Berechnung o.k. sonst:
S = 1 → ungültiger Parameter: `-X^(z/n)`
P = 1 → Überlauf
Unverändert: HL, FLO(DE)

BD6A BD8B BD8E FLO VGL SGN (FLO(HL)-FLO(DE)) → A

Eingaben: FLO(DE) und FLO(HL)
Ausgaben: A = –1/0/1 für FLO(HL)-FLO(DE) = negativ/null/positiv
Z = 1 → null
CY = 1 → negativ
Unverändert: BC, DE, HL, FLO(HL), FLO(DE)

Funktionen

Alle Funktionen (außer FLO VZW) werden über Polynomentwicklungen berechnet. Das ist leider nicht besonders schnell, läßt sich meist aber nicht sinnvoll umgehen. Es ist jedoch denkbar, für die eine oder andere Routine eine eigene zu installieren. Für die Wurzelberechnung gibt es beispielsweise viel schnellere Näherungsverfahren. Aber auch durch geschickte Abschätzung, wie viele Summanden (Polynome) für einen speziellen Eingabeparameter wirklich notwendig sind, lassen sich die meisten Routinen noch beschleunigen. Für Grafikanwendungen, bei denen in aller Regel die achte Stelle hinter dem Komma nicht mehr interessiert, läßt sich die Anzahl der benötigten Polynome oft erheblich vermindern. Es ist also auch möglich, die Routinen auf Kosten der Rechengenauigkeit zu beschleunigen.

Die Routinen FLO LOG NAT und FLO LOG DEC melden für das Argument 0 korrekt "Overflow" und liefern den Funktionswert 0. Möglicherweise wäre hier aber ein anderer Wert wünschenswert gewesen: –88.73 oder kleiner (bei FLO LOG NAT). Bedingt durch die Tatsache, daß der Schneider CPC mit seinem Fließkommaformat keine absolut beliebig kleinen Zahlen darstellen kann, sondern bereits etwa bei 0.3E–38 auf 0 runden muß, ergeben sich keine Funktionswerte, die kleiner als –88.73 sind. Das Argument 0 könnte also auch eine zwangsweise nach 0 gerundete kleine Zahl sein, deren Logarithmus normaler-

weise noch leicht darstellbar wäre. Der Wunsch nach der "kleinsten sinnvollen Zahl" entspringt also derselben Logik, die nach einer Division durch Null oder einem Überlauf die Möglichkeit bietet, mit der größtmöglichen Zahl weiterzurechnen.

BD6D BD8E BD91 FLO VZW −1 * FLO(HL) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
Unverändert: BC, DE, HL, IY

BD79 BD9A BD9D FLO SQR SQR (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 0 → negative Zahl
Unverändert: HL

BD7F BDA0 BDA3 FLO LOG NAT LOG (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 0 → Argument <= 0
Unverändert: HL

BD82 BDA3 BDA6 FLO LOG DEC LOG10 (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 0 → Argument <= 0
Unverändert: HL

BD85 BDA6 BDA9 FLO POT E E ^ FLO(HL) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 0 → Überlauf
Unverändert: HL

BD88 BDA9 BDAC FLO SIN SIN (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 0 → Argument zu groß
Unverändert: HL

BD8B BDAC BDAF FLO COS COS (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 0 → Argument zu groß
Unverändert: HL

BD8E BDAF BDB2 FLO TAN TAN (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
CY = 1 → o.k., sonst:
Z = 1 → Division durch Null
S = 1 → Argument zu groß
Unverändert: HL

BD91 BDB2 BDB5 FLO ARC TAN ARCTAN (FLO(HL)) → FLO(HL)

Eingaben: FLO(HL)
Ausgaben: FLO(HL)
Unverändert: HL

BD55 BD76 BD79 FLO 10^A FLO(HL) * 10^A → FLO(HL)

Eingaben: FLO(HL) und A
Ausgaben: FLO(HL)
CY = 0 → Überlauf
Unverändert: HL

BD67 ---- ---- FLO 2^A FLO(HL) * 2^A → FLO(HL)

Eingaben: FLO(HL), A

Ausgaben: FLO(HL)
CY = 0 → Überlauf
Unverändert: HL

BD70 BD91 BD94 FLO SGN SGN (FLO(HL)) → A

Eingaben: FLO(HL)
Ausgaben: A = −1/0/1 für FLO(HL) neg/null/pos
Z = 1 → null
C = 1 → negativ
Unverändert: BC, DE, HL, IY, FLO(HL)

Sonstiges

Die Kopierroutine für Fließkommazahlen ist mit Vorsicht zu genießen: Wie alle anderen FLO-Routinen auch, arbeitet sie nicht zwangsweise im RAM. Für den Vektor wurde von BASIC der Restart 5 (FIRM JUMP) gewählt. Wird FLO MOVE also von einem Maschinencode-Programm aufgerufen, so wird unten das ROM eingeblendet (nur für Lesezugriffe, also die Zahlenquelle interessant), ansonsten kann man von RAM ausgehen. Oft muß man aber auch eine Zahl aus dem unteren RAM-Bereich holen. Dann kann man diese Routine nicht benutzen.

BD3D BD5E BD61 FLO MOVE FLO(DE) → FLO(HL) und A:=exp.byte

Eingaben: FLO(DE) und (HL) = Fließkommaspeicher
Ausgaben: FLO(HL)
A = Exponentenbyte von FLO(DE) (Charakteristik)
CY = 1
Unverändert: BC, DE, HL, IX, IY, FLO(DE)

BD76 BD97 BD9A FLO PI PI → FLO(HL)

Eingaben: (HL) = Fließkommaspeicher
Ausgaben: FLO(HL)
CY = 1
Unverändert: BC, HL, IX, IY

BD73 BD94 BD97 FLO DEG/RAD A = 0 → RAD A>0 → DEG

Eingaben: A = 0 → Radiant
A > 0 → Degree
Ausgaben: keine
Unverändert: AF, BC, DE, HL, IX, IY

DIE INTEGER-ROUTINEN

Kleine ganze Zahlen im Bereich von –2^15 bis +2^15–1 (–32768 bis +32767) werden als "Integer" bezeichnet. Zur Darstellung negativer Zahlen wird die Komplementärdarstellung benutzt, da dieses Format auch vom Z80 unterstützt wird.

Die Parameterübergabe ist ähnlich streng strukturiert wie bei den Fließkommaroutinen, nur daß HL und DE hier nicht als Zeiger auf Fließkommavariablen dienen, sondern diese Zahlen direkt enthalten.

Operationen mit Vorzeichen

BDAC DD4F DD4A INT ADD VZ HL + DE → HL

Eingaben: HL und DE
Ausgaben: HL = HL+DE
CY = 0 → Überlauf
Unverändert: BC, DE, IX, IY

BDAF DD58 DD53 INT SUB VZ HL – DE → HL

Eingaben: HL und DE
Ausgaben: HL = HL–DE
CY = 0 → Überlauf
Unverändert: BC, DE, IX, IY

BDB2 DD57 DD52 INT SUB* VZ DE – HL → HL

Eingaben: HL und DE
Ausgaben: HL = DE–HL

Ausgaben: DE = alter Wert von HL
CY = 0 → Überlauf
Unverändert: BC, IX, IY

BDB5 DD60 DD5B INT MULT VZ HL * DE → HL

Eingaben: HL und DE
Ausgaben: HL = HL*DE
CY = 0 → Überlauf
Unverändert: DE, IX, IY

BDB8 DDA1 DD9C INT DIV VZ HL / DE → HL Rest DE

Eingaben: HL und DE
Ausgaben: HL = HL\DE
DE = ABS(HL MOD DE)
CY = 0 → Überlauf
Unverändert: IX, IY

BDBB DDA8 DDA3 INT MOD VZ HL / DE → DE Rest HL

Eingaben: HL und DE
Ausgaben: HL = HL MOD DE
DE = ABS(HL\DE)
CY = 0 → Überlauf
Unverändert: IX, IY

BDC4 DE07 DE02 INT VGL SGN (HL-DE) → A

Eingaben: HL und DE
Ausgaben: A = –1/0/1, wenn (HL-DE) = negativ/null/positiv
Z = 1 → HL = DE
CY = 1 → HL < DE
Unverändert: BC, DE, HL, IX, IY

Funktionen mit Vorzeichen

BDC7 DDF2 DDED INT VZW –1 * HL → HL

Eingaben: HL

Ausgaben: HL = –HL
CY = 0 → Überlauf (→ HL = –2^15)
Unverändert: BC, DE, IX, IY

BDCA DDFE DDF9 INT SGN SGN (HL) → A

Eingaben: HL
Ausgaben: A = –1/0/1 wenn HL negativ/null/positiv
Z = 1 → HL = 0
CY = 1 → HL < 0
Unverändert: BC, DE, IX, IY

Operationen ohne Vorzeichen

Der Darstellungsbereich dieser Routinen geht von 0 bis 2^16–1 (65535). Vorsicht bei der Multiplikation: Hier funktioniert das Carry-Flag mit umgekehrter Logik!

BDBE DD77 DD72 INT MULT HL * DE → HL

Eingaben: HL und DE
Ausgaben: HL = HL * DE
CY = 1 → Überlauf
Unverändert: BC, DE, IX, IY

BDC1 DDB0 DDAB INT DIV HL / DE → HL Rest DE

Eingaben: HL und DE
Ausgaben: HL = HL\DE
DE = HL MOD DE
CY = 0 → Überlauf. Auch Z = 1 → Division durch Null
Unverändert: BC, IX, IY

KONVERTIERUNGS-ROUTINEN

Insgesamt acht verschiedene Möglichkeiten werden bereitgehalten, um zwischen Integer und Fließkomma umzurechnen. Dabei wird für Integer-Zahlen nicht die übliche Komplementärdarstellung unterstützt, sondern nur eine getrennte Darstellung von Betrag und Vorzeichen. Für letzteres wird dabei immer das 7. Bit eines Registers (A oder B) benutzt.

BD46 BD67 BD6A ROUND FLO TO HLA ROUND(FLO(HL)) → HL, A=VZ

Eingaben: FLO(HL)
Ausgaben: HL = Absolutwert
Bit 7 von A = Vorzeichen
CY =0 → Überlauf
Unverändert: BC, DE, IY

BD40 BD61 BD64 KONV HLA TO FLO HL, A=VZ → FLO(DE)

Eingaben: HL = Absolutwert
Bit 7 von A = Vorzeichen
(DE) = Fließkommaspeicher
Ausgaben: HL = alter Wert von DE
FLO(HL)
Unverändert: BC, IX, IY

BD43 BD64 BD67 KONV LW TO FLO LW(HL), A=VZ → FLO(HL)

Eingaben: LW(HL) und Bit 7 von A = Vorzeichen
Ausgaben: FLO(HL)
Unverändert: BC, DE, HL, IY

BD49 BD6A BD6D ROUND FLO TO LW ROUND(FLO(HL)) → LW(HL), B=VZ

Eingaben: FLO(HL)
Ausgaben: LW(HL) = Absolutwert
Bit 7 von B = Vorzeichen
CY = 0 → Überlauf
Unverändert: DE, HL, IY

BD4C BD6D BD70 FIX FLO TO LW FIX(FLO(HL)) → LW(HL), B=VZ

Eingaben: FLO(HL)
Ausgaben: LW(HL) = Absolutwert
Bit 7 von B = Vorzeichen
Unverändert: DE, HL, IY

BD4F BD70 BD73 INT FLO TO LW INT(FLO(HL)) → LW(HL), B=VZ

Eingaben: FLO(HL)
Ausgaben: LW(HL) = Absolutwert
Bit 7 von B = Vorzeichen
CY = 0 → Überlauf
Unverändert: DE, HL, IY

BD94 BDB5 BDB8 KONV LW+C TO FLO LW(HL) * 256+C → FLO(HL)

Eingaben: LW(HL) und C
Ausgaben: FLO(HL) = LW(HL) * 256 + C
Unverändert: DE, HL, IY

BDA9 DD3C DD37 KONV HLB TO INT HL, B=VZ → HL

Eingaben: HL = Absolutwert
Bit 7 von B = Vorzeichen
Ausgaben: HL = Zahl in Komplementärdarstellung
CY = 0 → Überlauf
Unverändert: BC, DE, IX, IY

Dezimalwandlung

Die folgenden Routinen werden benötigt, wenn eine Fließkomma- oder Integer-Zahl ausgedruckt werden soll.

BD52 BD73 BD76 FLO PREPARE FLO(HL) → Parameter

Eingaben: FLO(HL)
Ausgaben: LW(HL) = normierte Mantisse
B = Vorzeichen der Mantisse
D = Vorzeichen des Exponenten
E = Exponent bzw. Kommaposition (je nach Darstellung)
C = Anzahl der signifikanten Mantissen-Bytes (nicht Ziffern!)
Unverändert: HL

BDA3 DD2F DD2A INT PREPARE VZ HL (Integer mit VZ)
→ Parameter

Eingaben: HL = Zahl in Komplementärdarstellung
Ausgaben: HL = Absolutwert
Bit 7 von B = Vorzeichen
C = 2 (Anzahl signifikanter Mantissen-Bytes)
E = 0 (Kommaposition)
Unverändert: IX, IY

BDA6 DD35 DD30 INT PREPARE HL (Integer ohne VZ)
→ Parameter

Eingaben: HL = positive Zahl (0 ... 2^16–1)
Ausgaben: B = 0 (Vorzeichen: positiv)
C = 2 (Anzahl signifikanter Mantissen-Bytes)
E = 0 (Kommaposition)
Unverändert: AF, D, IX, IY

Anhang A

Hardware-Basteleien

Der Schneider CPC erscheint über weite Strecken als ein Computer, bei dem gute Ideen in geradezu genialer Weise verwirklicht wurden.

Doch einige Details trüben das Bild: Sei es das fehlende 8. Bit für den Drucker, seien es die Beschränkungen des AMSDOS-Controllers oder die Tatsache, daß man den CPC zwar um 252 verschiedene ROMs, ohne Eingriffe am Gerät aber um keine einzige RAM-Bank erweitern kann.

So richtig unverständlich wird das Fehlen von manchen Funktionen dadurch, daß zu ihrer Implementierung nur ein Stückchen Draht oder ein einziges TTL-Gatter benötigt wird.

Bei den Bastelvorschlägen wurde darauf verzichtet, auf einige selbstverständliche Details wiederholt einzugehen. Es ist wohl klar, daß mit jedem Eingriff am Gerät ein eventueller Garantie-Anspruch erlischt, daß man vorher das Gehäuse öffnen muß, um an die Innereien heranzukommen, und daß man die Lötzeiten an einem IC-Beinchen möglichst kurz halten sollte.

Für die Lötarbeiten empfiehlt sich ein Lötkolben mit allerhöchstens 30 Watt, Elektronik-Lötzinn und bei trockenen Räumen und Kunststoff-Teppichboden eine regelmäßige statische Entladung des eigenen Körpers.

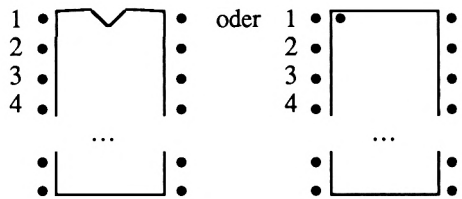
Das 8. Bit

Am Parallel-Port für den Anschluß eines Druckers nach dem Centronics-Standard fehlt das höchstwertige Datenbit. Sein Anschluß ist einfach auf Masse gelegt. Der Grund liegt darin, daß das achte Datenbit der CPU für das Strobe-Signal verwendet wird.

Bereits ein kurzes Stück isolierten Schaltdrahtes verhilft diesem Anschluß zum achten Bit: Trennen Sie die Leitung zu Pin 9 (Datenbit 7) des Drucker-Ports auf. Beim CPC 464 und 664 machen Sie das mit einem scharfen Messer, mit dem Sie die Leiterbahn, die zu diesem Pin führt, durchkratzen. Beim CPC

6128 durchtrennen Sie den entsprechenden Anschlußdraht der Buchse möglichst tief mit einem Seitenschneider. Verzählen Sie sich aber nicht, und beginnen Sie mit dem Zählen auf der richtigen Seite, sonst erwischen Sie den falschen Anschluß, und der Drucker funktioniert gar nicht mehr.

Suchen Sie jetzt die PIO auf der Platine (Bezeichnung: 8255). Hier interessiert Anschlußbeinchen 12. Das erhalten Sie, indem Sie einfach von Pin 1 im Gegenuhrzeigersinn weiterzählen. Pin 1 ist entweder mit einer Kerbe oder einem kleinen Loch wie folgt markiert:



Dieser Pin entspricht Bit 5 von Port B und dient als Ausgang für die Datenaufzeichnung auf Kassette. Es tut seiner Funktion aber keinen Abbruch, wenn er jetzt noch die Zusatzaufgabe bekommt, das achte Bit für den Drucker zu liefern. Es ist ja doch unmöglich, gleichzeitig zu drucken und Daten auf Kassette zu speichern.

Verbinden Sie jetzt diesen Pin 12 der PIO mit dem abgetrennten Anschluß 9 des Centronics-Ports mit einem kurzen Stück dünnen isolierten Drahtes, wofür Sie zweimal kurz zum Lötkolben greifen müssen. Bei den CPCs 464 und 664 mit ihren Platinensteckern müssen Sie darauf achten, daß so wenig Lötzinn wie möglich auf den Kontaktstreifen des Pin 9 selbst kommt. Der zur Verfügung stehende Platz ist hier leider sehr knapp.

Wenn das geschehen ist, sind auch schon alle Anpassungen erledigt, und Sie sollten wieder wie gewohnt drucken können. Wenn nicht, haben Sie etwas falsch gemacht. Schauen Sie sich Ihre Arbeit also noch einmal an.

Mit dieser Änderung allein können Sie noch keine 8 Bit breiten Daten zum Drucker senden, da die im Betriebssystem-ROM untergebrachte Treiberroutine von der Änderung keine Ahnung hat. Dazu muß noch an der Indirection &BDF1 IND MC WAIT PRINTER eingegriffen werden. Das folgende kleine Programm reicht beim CPC 464 bereits aus:

```
#BDF1 LD   BC,#F620 ; &F600 - Adresse von Port B der PIO
                    ; &0020 - Annahme: Bit 7 muss gesetzt werden
#BDF4 BIT  7,A
#BDF6 JR   NZ,#BDFA
```

```
#BDF8 LD   C,0      ; &0000 - Annahme falsch. Bit 7 muss null sein
#BDFA OUT  (C),C    ; Port B der PIO programmieren. Bit 7 ist gesetzt
#BDFC JP   #07F8    ; jetzt das Zeichen ganz normal ausdrucken
```

Damit wird direkt an der Stelle, an der die Indirection ist, die komplette Zusatzroutine untergebracht. Das geht, weil der Speicher danach bis &BDFF unbenutzt ist. Das ist beim CPC 664 und 6128 leider nicht mehr der Fall, weil hier an Adresse &BDF4 noch die Indirection IND SCAN KEYS eingefügt wurde. Hier muß man also ganz normal "patchen" und etwas RAM über HIMEN opfern. Die Adresse der normalen Behandlungsroutine für den abschließenden Sprung weicht dabei auch ab:

```
CPC 664 : &0825
CPC 6128: &0835
```

Ist die Zusatzroutine installiert, können Sonderzeichen und Grafik mit vollen acht Bits ausgegeben werden.

Externes RAM

Am Schneider CPC können bis zu 252 verschiedene ROMs angeschlossen werden. Zwar ist das ein ziemlicher "Overkill" im Vergleich zum tatsächlich zu erwartenden Bedarf, doch wäre das ja nicht so schlimm, wenn damit auch zusätzliche RAM-Bänke impliziert wären. Die könnte man, zumal beim CPC 464 und 664, schon eher gebrauchen.

Das ist leider nicht möglich, denn jeder Schreibbefehl der CPU geht automatisch an das eingebaute RAM. Der ROM-Status, der von der ULA in ROMEN und RAMRD umgesetzt wird, bezieht sich nur auf Lesebefehle. Unabhängig vom aktuellen ROM-Status wird bei jedem Schreibbefehl die Leitung MWE (Memory Write Enable) aktiviert, und die eingebauten RAMs lesen das Datenwort vom Datenbus der CPU.

Da hilft es auch nichts, daß der Eingang RAMDIS (RAM DISable) das Signal RAMRD unterdrücken kann. RAMDIS wirkt nur auf RAMRD und somit nur bei Lesebefehlen.

Man könnte diesen Zustand akzeptieren und trotzdem zusätzliches RAM geradeso wie zusätzliche ROMs anschließen. Bei jedem Schreibbefehl würden dann halt das eingebaute und das externe RAM gleichzeitig beschrieben. Wenn man sich an den ROM-Standard hält (zusätzliche RAM-Blocks also nur im obersten Speicherviertel), würden diese Daten normalerweise "nur" im Bildschirmspeicher als bunte Farbmuster erscheinen.

Als Ausweg bietet sich hier an, die Funktion von RAMDIS einfach auch auf Speicher-Schreibbefehle auszudehnen. Dazu müßte in die Leitung zwischen dem Anschluß MWE der ULA und WE der RAM-Bausteine ein "Schalter" eingesetzt werden, der von RAMDIS gesteuert wird. Einer möglichen Lösung dieses Problems ist man schon sehr nahe, wenn man erkennt, daß RAMDIS auf den Ausgang RAMRD genau in dieser Weise wirkt: RAMRD wird über ein Oder-Gatter mit RAMDIS verknüpft. Der Ausgang dieses Gatters steuert erst die Ausgabe der RAM-Daten auf den Datenbus der CPU.

Logikdiagramm für ein Oder-Gatter:

RAMDIS	RAMRD	Ausgabe	Bemerkung
0	0	0	aktiv
0	1	1	passiv
1	0	1	passiv
1	1	1	passiv

Eine 1 am Eingang RAMDIS kann das Durchschalten von RAMRD durch den Oder-Schalter verhindern.

Es genügt also der Einbau eines einzigen Gatters, um die Funktion von RAMDIS auch auf Schreibbefehle auszudehnen, womit dann einer externen RAM-Erweiterung nichts mehr im Wege steht. Man muß nur ein solches Oder-Gatter auch zwischen MWE (ULA) und WE (RAMs) einfügen.

Bastelanleitung für RD- und WR-wirksames RAMDIS

Benötigt werden ein Lötkolben, Lötzinn, etwas isolierter Schaltdraht, ein Widerstand mit 2.2 Kiloohm und ein IC vom Typ 74LS32.

Zuerst muß die Verbindung zum Ausgang MWE (Memory Write Enable) der ULA aufgetrennt werden. Das ist beim CPC 464 und 664 der Pin 5 der ULA. Beim CPC 6128 ist es Pin 33. Da die ULAs in allen CPC-Typen gesockelt sind, geht das recht leicht, wenn Sie das IC aus dem Sockel heraushebeln, das Beinchen nach außen abbiegen und dann die ULA wieder einsetzen. (Bauen Sie vorher sicherheitshalber elektrostatische Aufladungen ab, indem Sie einen größeren Metallgegenstand berühren.)

Suchen Sie für den 74LS32 einen Platz in der Nähe des operierten Pins der ULA, wo sie ihn mit Doppelklebeband o.ä. leicht befestigen können. Sie können dazu alle Beinchen des ICs waagerecht hochbiegen und sogar alle abschneiden bis auf folgende: 1, 2, 3, 7 und 14.

Verbinden Sie jetzt Pin 1 des 74LS32 mit Pin 5 (33) der ULA, Pin 2 des 74LS32 mit Anschluß 45 des Expansion-Ports und Pin 3 des 74LS32 mit Pin 5 (33) des Sockels der ULA.

Jetzt fehlt nur noch die Stromversorgung für das IC, für die Sie Verbindungen von Pin 7 und 14 zu Pin 7 und 14 eines benachbarten, ähnlichen ICs herstellen. "Ähnlich" in diesem Sinne sind auf jeden Fall alle 14poligen ICs der Serie 74LS... Sehr empfehlenswert ist beim Schneider CPC 464 das IC 7400, das sich ganz rechts auf der Platine befindet. Sie können das IC 74LS32 auch auf ein solches IC draufsetzen und alle Beinchen bis auf 7 und 14 abbiegen und dann diese Pins (7 und 14) direkt an dieselben Pins des anderen ICs anlöten.

Wenn Sie jetzt noch den Widerstand von Pin 2 nach Masse (Pin 7) einlöten, sind Sie schon fertig. Vorausgesetzt, Sie haben nichts falsch gemacht und auch kein IC "geschossen", läuft Ihr Schneider CPC wieder, ohne daß Sie eine Änderung bemerken. Sie können jetzt den Computer um praktisch beliebig viele RAM-Bänke erweitern, ohne wieder Eingriffe im Gerät machen zu müssen. Da das eingebaute RAM jetzt bei Schreib- und Lesezugriffen ausgeblendet werden kann, können Sie zusätzliche RAM-Karten am Systembus anschließen.

Ein RESET-Knopf

Normalerweise kann der Schneider CPC jederzeit durch Drücken der Tasten CTRL, SHIFT und ESC zurückgesetzt werden. Ein solcher Eingriff ins laufende Programm läßt sich manchmal nicht vermeiden. Diese Tastenkombination zwingt den Prozessor dabei nicht, einen RESET durchzuführen. Der Key Manager überprüft aber bei jeder normalen Tastaturabfrage, ob diese Tastenkombination gedrückt ist. Wenn ja, wird aus der Interrupt-Routine heraus der Rechner neu gestartet.

Vor allem Maschinencode-Programme haben aber bisweilen die unangenehme Eigenschaft, so "gründlich" abzustürzen, daß auch die Tastaturabfrage nicht mehr in gewohnter Weise durchgeführt wird. Dann helfen auch diese drei Tasten nicht.

Aus diesem Zustand läßt der Rechner sich nur noch durch Ausschalten erlösen. Schaltet man aber den Monitor aus und wieder ein, so leidet unweigerlich dessen Elektronik. Außerdem muß man sich mit dem Wiedereinschalten immer etwas Zeit lassen. Fünf Sekunden und auch mehr erscheinen so manchem CPC als angemessen. Ohne diese Karenzzeit erzeugt die eingebaute "Power-On-RESET-Schaltung" keinen korrekten Impuls.

Als Ausweg bietet sich hier ein kleiner Tastschalter an, der zwei Kontakte zusammenschaltet, wenn er gedrückt wird.

Der Schalter wird am besten an der Rückseite des Computers angebracht, wo er nicht stört, jederzeit leicht erreichbar ist, aber keinesfalls unbeabsichtigt ausgelöst werden kann.

Verbinden Sie einen Anschluß des Schalters mit Masse, beispielsweise am Pin 2 des Expansion-Ports, und den anderen Anschluß mit dem Eingang BUS RESET. Das ist Pin 40 (wie Pin 2) auf der Unterseite der Platine. Das ist schon alles. Der Rechner sollte jetzt wieder wie gewohnt laufen. Bei jedem Druck auf den Taster muß er einen Kaltstart durchführen.

Dieser Kaltstart hat vor CTRL-SHIFT-ESC sogar noch den Vorteil, daß damit der Floppy-Controller auch dann zurückgesetzt wird, wenn man ihn scherzhafterweise auf DMA-Betrieb eingestellt hat.

Anhang B

Systemspeicher im CPC

Das RAM des Betriebssystems

RAM des Kernel

CPC 464	CPC 664	CPC 6128	Bedeutung der Speicherstelle(n)
b100,b101	b82d,b82e	b82d,b82e	Start der Asynchronous Pending Queue
b102,b103	b82f,b830	b82f,b830	letzter Block in der Asyn. Pend. Queue
b104	b831	b831	Flags für Queue-Bearbeitungen
b105,b106	b832,b833	b832,b833	Zw.speicher für SP bei Queue-Bearbeitungen
b107-b186	b834-b8b3	b834-b8b3	Privat-Stack bei Queue-Bearbeitung
b187-b18a	b8b4-b8b7	b8b4-b8b7	TIME-Speicher
b18b	b8b8	b8b8	Sperrbyte gegen TIME-Überlauf
b18c,b18d	b8b9,b8ba	b8b9,b8ba	Start der Frame Flyback Chain
b18e,b18f	b8bb,b8bc	b8bb,b8bc	Start der Fast Ticker Chain
b190,b191	b8bd,b8be	b8bd,b8be	Start der Ticker Chain
b192	b8bf	b8bf	1/6-Zählbyte für Ticker
b193,b194	b8c0,b8c1	b8c0,b8c1	Start der Synchronous Pending Queue
b195	b8c2	b8c2	aktuelle Synchronous Event-Priorität
b196-b1a5	b8c3-b8d2	b8c3-b8d2	Puffer für RSX-Name → KL FIND COMMAND
b1a6,b1a7	b8d3,b8d4	b8d3,b8d4	Start der External Command Chain (RSX)
----	----	b8d5	aktuelle RAM-Konfiguration
b1a8	b8d5	b8d6	aktuelle ROM-Konfiguration
b1a9,b1aa	b8d6,b8d7	b8d7,b8d8	Startadr. des lfd. Vordergrund-Programms
b1ab	b8d8	b8d9	& dessen ROM-Konfiguration (→ SIDE CALL)
b1ac-b1b9	b8d9-b8f8	b8da-b8f9	IY-Speicher für die Hintergrund-ROMs CPC 464: 1–7 / CPC664/6128: 0–15
b1ba-b1c7	b8f9-b8ff	b8fa-b8ff	{unbenutzt}

RAM des Machine Pack

CPC 664/6128	Bedeutung der Speicherstelle(n)
b804-b82c	Drucker-Übersetzungstabelle

RAM des Screen Pack

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b1c8	b7c3	Bildschirmmodus
b1c9,b1ca	b7c4,b7c5	Scroll-Offset der RAM-Zeilen
b1cb	b7c6	MSB des Bildschirmstarts (&00, &40, &80 oder &C0)
b1cc-b1ce	b7c7-b7c9	Indirection zum Punkte-Plotten: force/and/or/xor
b1cf-b1d6	----	Masken-Bytes für Pixel im Byte, je nach Mode
----	b7ca-b7d1	{unbenutzt}
b1d7	b7d2	Blinkperiode Farbsatz 1
b1d8	b7d3	Blinkperiode Farbsatz 0
b1d9-b1e9	b7d4-b7e4	Paletten-Farbnummern für Border&Inks Farbsatz 1
b1eb-b1fa	b7e5-b7f5	Paletten-Farbnummern für Border&Inks Farbsatz 0
b1fb	b7f6	Flag für aktuellen Farbsatz
b1fc	b7f7	Flag für neu zugeordnete Farben in der Tabelle
b1fd	b7f8	Count Down für aktuelle Blink-Periode
b1fe-b206	b7f9-b801	Frame Flyback Block für Farbblinken
b207,b208	b802,b803	diverse Speicher für Grafik-Routinen
b209-b20b	----	{unbenutzt}

RAM der Text-VDU

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b20c	b6b5	aktuelles Textfenster
b20d-b21b	b6b6-b6c3	Parameter für Fenster 0
b21c-b22a	b6c4-b6d1	Parameter für Fenster 1
b22b-b239	b6b2-b6df	Parameter für Fenster 2
b23a-b248	b6e0-b6ed	Parameter für Fenster 3
b249-b257	b6ee-b6fb	Parameter für Fenster 4
b258-b266	b6fa-b709	Parameter für Fenster 5
b267-b275	b70a-b717	Parameter für Fenster 6
b276-b284	b718-b725	Parameter für Fenster 7
b285-b293	b726-b733	Parameter für das aktuelle Textfenster
b285	b726	Cursor-Zeile (links oben = (0,0))
b286	b727	Cursor-Spalte (links oben = (0,0))
b287	b728	Flag für Hardware-Scroll möglich (0-HW/&FF-SW)
b288	b729	Fenstergrenze oben
b289	b72a	Fenstergrenze links
b28a	b72b	Fenstergrenze unten
b28b	b72c	Fenstergrenze rechts
b28c	b72d	Zähl-Byte für Scrolls hoch/runter
b28d	----	Cursor: b0= 0-enabled/1-disabled/b1= 0-on/1-off
b28e	----	Textausgabe: 0 – disabled / <>0 – enabled
----	b72e	Cursor: b0= 0-enabled/1-disabled/b1= 0-on/1-off und Textausgabe: b7= 0-enabled/1-disabled
b28f	b72f	Farb-Byte (encoded INK) für PEN

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b290	b730	Farb-Byte (encoded INK) für PAPER
b291,2	b731,2	Routinenadresse entsprechend Hintergrund-Modus
b293	b733	Text-at-graphics-Flag: 0 – TAGOFF / <>0 – TAG
b294	b734	CHR$()-Nummer der ersten Zeichenmatrix im RAM
b295	b735	Flag: 0 – keine / &FF – Matrizen im RAM
b296,b297	b736,b737	Startadresse der Zeichenmatrizen im RAM
b298-b2b7	b738-b757	Puffer für expandierte Zeichenmatrix
b2b8	b758	Anzahl Zeichen im Controlcode-Puffer (<>0 – Controlcode wartet auf Parameter)
b2b9-b2c2	b759-b762	Controlcode-Puffer
b2c3-b322	b763-b7c2	Controlcode-Tabelle (Anz. Arg. & Routinen-Adr.)
b323-b327	----	{unbenutzt}

RAM der Graphics-VDU

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b328,b329	b693,b694	Origin: X-Koordinate
b32a,b32b	b695,b696	Origin: Y-Koordinate
b32c,b32d	b697,b698	Grafik-Cursor: X-Koordinate
b32e,b32f	b699,b69a	Grafik-Cursor: Y-Koordinate
b330,b331	b69b,b69c	Grafik-Fenstergrenze: links
b332,b333	b69d,b69e	Grafik-Fenstergrenze: rechts
b334,b335	b69f,b6a0	Grafik-Fenstergrenze: oben
b336,b337	b6a1,b6a2	Grafik-Fenstergrenze: unten
b338	b6a3	Farbbyte (encoded INK) für Vordergrund-Pixel
b339	b6a4	Farbbyte (encoded INK) für Hintergrund-Pixel
b33a-b346	b6a5-b6b1	Zwischenspeicher für diverse Aufgaben (Print Char. bei TAG, DRAW Linie, FILL)
----	b6b2	Flag für Erster-Punkt-Option
----	b6b3	Linienmaske
----	b6b4	Hintergrund-Modus: 0 – opaque / &FF – transpar.
b347-b34b	----	{unbenutzt}

RAM des Keyboard Managers

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b34c-b39b	b496-b4e5	Tasten-Übersetzungstabelle für Taste SOLO
b39c-b3eb	b4e6-b535	Tasten-Übersetzungstabelle für Taste mit SHIFT
b3ec-b43b	b536-b585	Tasten-Übersetzungstabelle für Taste mit CTRL
b43c-b445	b586-b58f	Tasten-Repeat-Tabelle
b446-b4dd	b590-b627	Expansion-String-Puffer
b4de	b628	Zähler im Expansion-String
b4df	b629	Nummer des aktuellen Erweiterungszchn. (wenn <>0)

RAM des Keyboard-Managers

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b4e0	b62a	Puffer für put back character
b4e1,b4e2	b62b,b62c	Zeiger auf Expansion-String-Puffer
b4e3,b4e4	b62d,b62e	Zeiger auf dessen Ende
b4e5,b4e6	b62f,b630	Zeiger auf den noch freien Bereich darin
b4e7	b631	b7= 0 – kein SHIFT LOCK / =1 – SHIFT LOCK
b4e8	b632	b7= 0 – kein CAPS LOCK / =1 – CAPS LOCK
b4e9	b633	erste Verzögerungszeit beim "Repeaten"
b4ea	b634	Zeit für die Wiederholverzögerung dabei
b4eb-b4f4	b635-b63e	Tabelle für aktuell gedrückte Tasten
b4f5-b4fe	b63f-b648	Zwischentabelle dafür
b4ff-b508	b649-b652	Zwischentabelle dafür
b509	b653	Count Down für Repeat
b50a,b50b	b654,b655	aktuelle Taste (physikalische Informationen)
b50c	b656	Flag für Break-Mechanismus: <>0 – scharf
b50d-b513	b657-b65d	Break-Event-Block
b514-b53b	b65e-b685	Warteschlange für gedrückte Tasten (physikalische Informationen)
b53c-b540	b686-b68a	Parameter zur Verwaltung der Warteschlange
b541,b542	b68b,b68c	Zeiger auf Tasten-Übersetzungstabelle SOLO
b543,b544	b68d,b68e	Zeiger auf Tasten-Übersetzungstabelle mit SHIFT
b545,b546	b68f,b690	Zeiger auf Tasten-Übersetzungstabelle mit CTRL
b547,b548	b691,b692	Zeiger auf Tasten-Repeat-Tabelle
b549-b54f	----	{unbenutzt}

RAM des Sound Manager

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b550	----	Zwischenspeicher für restl. Kanalaktivität
b551	b1ed	alte Kanalaktivität (für SOUND CONTINUE)
b552	b1ee	aktuelle Kanalaktivität (b0/1/2 = Kanal A/B/C)
b553	b1ef	1/3-Zähl-Byte für Sound-Chain
b554	b1f0	Flag für "Kanal zu bearbeiten"
b555-b55b	b1f1-b1f7	Event-Block für die Tonausgabe
b55c-b59a	b1f8-b236	Parameterblock für Kanal A (u.a. Sound-Queue)
b59b-b5d9	b237-b275	Parameterblock für Kanal B (u.a. Sound-Queue)
b5da-b618	b276-b2b4	Parameterblock für Kanal C (u.a. Sound-Queue)
b619	b2b5	Byte für Kontroll-Register des PSG
b61a-b709	b2b6-b3a5	Lautstärke-Hüllkurven 1 bis 15
b70a-b7f9	b3a6-b495	Frequenz-Hüllkurven 1 bis 15
b7fa-b7ff	----	{unbenutzt}

RAM des Cassette-Managers

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b800	b118	0 – Message on / &FF – Message off
b801	b119	0 – Meldung in einem Stück / &FF zerteilt
b802	b11a	Status der Input-Datei
b803,b804	b11b,b11c	Adresse des 2k-Input-Puffers
b805,b806	b11d,b11e	Zeiger im Input-Puffer
b807-b846	b11f-b15e	Header-Puffer für Input
b847	b15f	Status der Output-Datei
b848,b849	b160,b161	Adresse des 2k-Output-Puffers
b84a,b84b	b162,b163	Zeiger im Output-Puffer
b84c-b88b	b164-b1a3	Header-Puffer für Output
b88c-b8cb	b1a4-b1e3	Puffer für neu gelesenen Header
b8cc	b1e4	b0: Eingabe 0 – nicht aktiv / 1 – aktiv b1: Ausgabe 0 – nicht aktiv / 1 – aktiv
b8cd	b1e5	Synchronisationszeichen
b8ce-b8d0	b1e6-b1e8	diverse Zwischenspeicher beim Lesen/Schreiben
b8d1	b1e9	Präkompensation
b8d2	b1ea	Speichergeschwindigkeit
b8d3,b8d4	b1eb,b1ec	CRC-Prüfwort
b8d5-b8db	----	{unbenutzt}

RAM des Zeileneditors

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b8dc	b114	Cursor-Copy-Cursor-Flag
b8dd	b115	Insert-Flag
b8de,b8df	b116,b117	Koordinaten des Copy-Cursors
b8e0-b8e3	----	{unbenutzt}

RAM des Floating Point Pack

CPC 464	CPC 664/6128	Bedeutung der Speicherstelle(n)
b8e4-b8e7	b100-b103	RND-Zahl (LW oder FLO normalisiert ohne ↵ Exponent)
b8e8-b8f6	b104-b112	3 Zwischenspeicher für Fließkommazahlen
b8f7	b113	Flag für 0 – RAD / <>0 – DEG
b8f8-b8ff	----	{unbenutzt}

Das RAM des BASIC-Interpreters

CPC 464	CPC 664/128	Bedeutung der Speicherstelle(n)
ac00	ac00	Flag für Space-Unterdrückung beim Tokenisieren
ac01-ac1b	----	BASIC-Indirections:
ac01	----	Eingabeschleife
ac04	----	Fehlerausgabe
ac07	----	Befehlsausführung
ac0a	----	Funktionsauswertung
ac0d	----	Operandenauswertung
ac10	----	ASCII-Wort tokenisieren
ac13	----	Token zurück nach ASCII wandeln
ac16	----	Schlüsselwort tokenisieren
ac19	----	Ausführungsroutine zu einem Token suchen
ac1c	ac01	Flag für AUTO
ac1d,ac1e	ac02,ac03	aktuelle Zeilennummer für AUTO
ac1f,ac20	ac04,ac05	Schrittweite für AUTO
ac21	ac06	aktueller Ausgabekanal (Stream)
ac22	ac07	aktueller Eingabekanal
ac23	ac08	aktuelle X-Position auf dem Drucker
ac24	ac09	WIDTH
ac25	ac0a	aktuelle X-Position in der Ausgabedatei
---	ac0b	Flag für ON BREAK CONT (0=aktiv)
ac26	ac0c	Flag für NEXT-Behandlungsroutine
ac27-ac2b	ac0d-ac11	Speicher für Startwert in FOR-NEXT-Schleife
ac2c,ac2d	ac12,ac13	Zeiger hinter zugehöriges NEXT
ac2e,ac2f	ac14,ac15	Zeiger auf Zeile mit zugehörigem WEND
ac30	ac16	Flags für die Bearbeitung synchroner Events
ac31-ac35	ac17-ac1b	Parameterblock für ON BREAK GOSUB:
ac31	ac17	alte Priorität (A-Register nach KL NEXT SYNC)
ac32,33	ac18,19	BASIC-Rücksprungadresse (PC im BASIC-Programm)
ac34,35	ac1a,1b	Adresse des BASIC-Unterprogramms
ac36,ac37	ac1c,ac1d	Zeiger auf Routinenadresse im Break-Event-Block
ac38-ac43	ac1e-ac29	ON SQ(1) GOSUB
ac38-3e	ac1e-24	Event-Block
ac3f-43	ac25-29	Parameterblock (wie bei ON BREAK GOSUB)
ac44-ac4f	ac2a-ac35	ON SQ(2) GOSUB
ac50-ac5b	ac36-ac41	ON SQ(4) GOSUB
ac5c-ac6d	ac42-ac53	EVERY/AFTER ,0 GOSUB
ac5c-68	ac42-4e	Ticker Chain Block
ac69-6d	ac4f-53	Parameterblock (wie bei ON BREAK GOSUB)
ac6e-ac7f	ac54-ac65	EVERY/AFTER ,1 GOSUB
ac80-ac91	ac66-ac77	EVERY/AFTER ,2 GOSUB
ac92-aca3	ac78-ac89	EVERY/AFTER ,3 GOSUB
aca4-ada5	ac8a-ad8b	*** ASCII-Puffer *** (INPUT, LIST)

CPC 464	CPC 664/128	Bedeutung der Speicherstelle(n)
ada6,ada7	ad8c,ad8d	Zeilenadresse des letzten Fehlers für ERL
ada8,ada9	ad8e,ad8f	Statement-Adresse des letzten Fehlers
adaa	ad90	Nummer des letzten Fehlers für ERR
----	ad91	Fehlernummer für DERR
adab,adac	ad92,ad93	Statement-Adresse nach BREAK für CONT
adad,adae	ad94,ad95	Zeilenadresse nach BREAK für CONT
adaf,adb0	ad96,ad97	Adresse des BASIC-Programms für ON ERROR GOTO
adb1	ad98	Flag für ON ERROR (&FF = im On-Error-Pfad)
adb2-adba	ad99-ada1	Puffer für SOUND-Parameter
adbb-adca	ada2-adb1	Puffer für ENV- und ENT-Parameter
adcb-adcf	adb2-adb6	Zwischenspeicher beim Potenzieren
add0-ae03	adb7-adea	Start-Pointer der Chains der normalen Variablen (26 Stück für jeden Variablentyp)
ae04,ae05	adeb,adec	Start-Pointer der Chain der DEF FN
ae06,ae07	aded,adee	Start-Pointer der Chain der Real-Variablenfelder
ae08,ae09	adef,adf0	Start-Pointer der Chain der Integer-Variablenfelder
ae0a,ae0b	adf1,adf2	Start-Pointer der Chain der String-Variablenfelder
ae0c-ae25	adf3-ae0c	Default-Variablentyp: DEFINT, DEFREAL, DEFSTR (26 Stück für jeden Anfangsbuchstaben)
ae26	ae0d	Flag für ad-hoc-Dimensionierung von Feldern
ae27-ae2c	ae0e-ae13	Zeiger beim Auswerten von Ausdrücken
ae2d	ae14	Flag für CR/LF nach INPUT
ae2e,ae2f	ae15,ae16	Zeilenadresse des DATA-Zeigers
ae30,ae31	ae17,ae18	DATA-Zeiger
ae32,ae33	ae19,ae1a	BASIC-Stackpointer zum Statement-Anfang
ae34,ae35	ae1b,ae1c	Adresse des aktuellen Statements
ae36,ae37	ae1d,ae1e	Adresse der aktuellen BASIC-Zeile
ae38	ae1f	Trace-Flag: 0-TROFF / &FF-TRON
ae39	ae20	Flag beim Tokenisieren
ae3a	ae21	Flag: 0 – keine Zeilenadressen im Programm; Programmtext ist ortsunabhängig
ae3b-ae3e	ae22-ae25	Parameter fuer DELETE
ae3f,ae40	ae26,ae27	Startadresse beim Laden von Programmen
ae41	ae28	Flag für CHAIN/CHAIN MERGE
ae42	ae29	Speicher für File-Typ
ae43,ae44	ae2a,ae2b	File-Länge
ae45	ae2c	Flag für geschütztes BASIC-Programm, wenn &FF
ae46-ae6d	ae2d-ae51	Puffer für Zahlenwandlung
ae6e-ae71	ae52-ae54	diverse Speicher bei der Zahlenwandlung
ae72-ae74	ae55-ae57	Far Address für CALL- oder RSX-Aufruf
ae75,ae76	ae58,ae59	Speicher für BASIC-Programmzeiger bei CALL/RSX
ae77,ae78	ae5a,ae5b	Speicher für den SP der CPU bei CALL/RSX
ae79	ae5c	ZONE

CPC 464	CPC 664/128	Bedeutung der Speicherstelle(n)
ae7a	ae5d	Flag für Ende des Format-Strings bei PRINT USING
ae7b,ae7c	ae5e,ae5f	Systemspeicher für HIMEM
ae7d,ae7e	ae60,ae61	Ende des BASIC-RAMs nach KL ROM WALK
ae7f,ae80	ae62,ae63	Start des BASIC-RAMs nach KL ROM WALK
ae81,ae82	ae64,ae65	Start des BASIC-Programms
ae83,ae84	ae66,ae67	Ende des BASIC-Programms
ae85,ae86	ae68,ae69	Start des Variablenbereichs
ae87,ae88	ae6a,ae6b	Start des Bereichs der Felder
ae89,ae8a	ae6c,ae6d	Ende der Felder
----	ae6e	Flag für geschützten Variablenbereich
ae8b-b08a	ae6f-b06e	*** BASIC-Stack ***
b08b,b08c	b06f,b070	Stackpointer im BASIC-Stack
b08d,b08e	b071,b072	Anfang der Strings
b08f,b090	b073,b074	Ende der Strings
b091	b075	Flag für den I/O-Puffer: b0=1 – Input aktiv / b1=1 – Output aktiv / b2=1 – Puffer reserviert
b092,b093	b076,b077	Zeiger auf (CPC 464: vor) I/O-Puffer
b094-b099	b078-b07b	Zwischenspeicher bei Änderungen von HIMEM
b09a,b09b	b07c,b07d	Stackpointer im String-Descriptor-Stack
b09c-b0b9	b07e-b09b	String-Descriptor-Stack
b0ba-b0bc	b09c-b09e	Puffer für einen String-Descriptor
b0bd-b0c0	----	Speicher bei einer Garbage Collection
b0c1	b09f	Typ des BASIC-Akku: Real, String oder Integer
b0c2-b0c6	b0a0-b0a4	Akku bei der Auswertung von Ausdrücken (Integer, Real oder Zeiger auf String-Descriptor)
b0c7-b0ff	b0a5-b0ff	{unbenutzt}

Das RAM des AMSDOS-Disketten-Controllers

DIE ORTSFESTEN SYSTEMSPEICHER

CPC 464/664/6128	Bedeutung der Speicherstelle(n)
be40,be41	Adresse des DPH (DisK Parameter Header), Laufwerk A
be42,be43	Adresse des DPB (Disk Parameter Block), Laufwerk A
be44,be45	Wartezeit nach Starten des Motors
be46,be47	Wartezeit bis zum Stoppen des Motors nach dem letzten Zugriff
be48	Verzögerungszeit beim Formatieren
be48,be4a	Verzögerungszeit
be4b	Anzahl Bytes aus der Resultphase des FDC

CPC 464/664/6128	Bedeutung der Speicherstelle(n)
be4c-be52	Puffer für Bytes aus der Resultphase
be53-be5d	diverse Speicher für Sektorzugriff
be5e	Flag für Sektor lesen/schreiben
be5f	Flag für Motor an/aus
be60,be61	Zeiger auf I/O-Puffer für einen Record
be62,be63	Zeiger auf I/O-Puffer für einen Sektor
be64,be65	Zwischenspeicher für SP
be66	Retry Count (max. Anzahl für Leseversuche)
be67-be73	Ticker Chain Block für Motor-Aus-Event
be74	Speicher für angewählte Spur
be75	Speicher für Befehlsbyte zum FDC
be76,be77	Zeiger auf I/O-Puffer für einen Sektor
be78	Flag für Message on/off
be79-be7c	{unbenutzt}
be7d,be7e	Puffer für IY = Adresse des dynamisch zugeteilten Speichers
be7f-be81	Indirection für alle von AMSDOS gepatchten Vektoren

DAS DYNAMISCH ZUGETEILTE RAM

CPC 464/664/6128	Bedeutung der Speicherstelle(n)
a700	momentan angewähltes Laufwerk
a701	aktuelle USER-Nummer
a702	aktives Laufwerk
a703,a704	Zeiger auf den Disk Parameter Header des aktiven Laufwerks
a705	Flag für I/O-Datei auf aktivem Laufwerk offen
a706,a707	Zwischenspeicher für SP
a708-a72b	erweiterter File Control Block für OPENIN:
a708	Flag: &FF – keine Datei eröffnet, sonst 0/1 = Drive
a709-a728	aktuell benötigter Directory-Extent
a709	USER
a70a-a714	File-Name und Extension
a715	Nummer dieses Extents
a716,a717	{unbenutzt}
a718	Anzahl Records in diesem Extent
a719-a728	Block-Belegungstabelle
a729-a72b	Anzahl bisher gelesener Records
a72c-a74f	erweiterter File Control Block für OPENOUT:
a72c	Flag: &FF – keine Datei eröffnet, sonst 0/1 = Drive
a72d-a74c	aktuell benötigter Directory-Extent
a72d	USER
a72e-a738	File-Name und Extension
a739	Nummer dieses Extents

CPC 464/664/6128	Bedeutung der Speicherstelle(n)
a73a,a73b	{unbenutzt}
a73c	Anzahl Records in diesem Extent
a73d-a74c	Block-Belegungstabelle
a74d-a74f	Anzahl bisher geschriebener Records
a750-a799	erweiterter Datei-Header für OPENIN:
a750	Flag für zeichenweise (1) lesen oder im Block (2)
a751,a752	Adresse des 2k-Input-Puffers
a753,a754	Lesezeiger im Input-Puffer
a755-a794	Datei-Header ähnlich dem Cassette Manager
a755	USER
a756-a764	File-Name und Extension, mit Null-Bytes aufgefüllt
a765,a766	ohne Bedeutung
a767	Dateityp
a768,a769	ohne Bedeutung
a76a,a76b	Originallage der Datei beim Saven
a76c	ohne Bedeutung
a76d,a76e	logische Dateilänge
a76f,a770	Startadresse für Maschinencode-Programme
a771-a794	User field: unbenutzt, kann vom Anwender beschrieben werden
a795-a797	Zähler über gelesene Bytes
a798,a799	Prüfsumme über den Datei-Header (a755 bis a797)
a79a-a7e3	erweiterter Datei-Header für OPENOUT:
a79a	Flag für zeichenweise (1) oder im Block (2)
a79b,a79c	Adresse des 2k-Output-Puffers
a79d,a79e	Schreibzeiger im Output-Puffer
a79f-a7de	Datei-Header ähnlich dem Cassette Manager
a79f	USER
a7a0-a7ae	File-Name und Extension, mit Null-Bytes aufgefüllt
a7af,a7b0	ohne Bedeutung
a7b1	Dateityp
a7b2,a7b3	ohne Bedeutung
a7b4,a7b5	Originallage der Datei beim Saven
a7b6	ohne Bedeutung
a7b7,a7b8	logische Dateilänge
a7b9,a7ba	Startadresse für Maschinencode-Programme
a7bb-a7de	User field: unbenutzt, kann vom Anwender beschrieben werden
a7df-a7e1	Zähler über gelesene Bytes
a7e2,a7e3	Prüfsumme über den Datei-Header (a79f bis a7e1)
a7e4-a863	Record-Puffer (auch zum Expandieren von Dateinamen benutzt)
a864-a88a	Puffer für die Kopie der 13 gepatchten CAS-Vektoren
a88b-a88d	Far Address für RST 3 ins AMSDOS-ROM
a88e-a88f	{unbenutzt}

CPC 464/664/6128	Bedeutung der Speicherstelle(n)
a890-a8cf	Extended Disk Parameter Block für Laufwerk A
a890,a891	Records pro Track
a892	Block-Shift
a893	Block-Maske
a894	Extent-Maske
a895,a896	höchste benutzbare Blocknummer
a897,a898	Anzahl Extents im Directory –1
a899,a89a	Directory-Größe, jedes 1-Bit ⇔ 1 Block
a89b,a89c	Anzahl zu prüfender Extents für Disc-Change-Erkennung
a89d,a89e	Anzahl belegter Systemspuren
a89f	Nummer des ersten Sektors einer Spur
a8a0	Sektoren pro Spur
a8a1	Länge der Gap3 beim Sektor-Lesen/Schreiben
a8a2	Länge der Gap3 beim Formatieren
a8a3	Füll-Byte beim Formatieren
a8a4	Sektorlänge (in der FDC-Codierung)
a8a5	Records pro Sektor
a8a6	aktuelle Spurnummer
a8a7	Flag für Recalibrate vor Spursuchen
a8a8	Flag für Login vor jedem Diskettenzugriff
a8a9-a8b8	Puffer für Checksummen
a8b9-a8cf	23 Bytes für die Block-Belegungstabelle
a8d0-a90f	Extended Disk Parameter Block für Laufwerk B
a8d0,a8d1	Records pro Track
a8d2	Block Shift
a8d3	Block-Maske
a8d4	Extent-Maske
a8d5,a8d6	höchste benutzbare Blocknummer
a8d7,a8d8	Anzahl Extents im Directory –1
a8d9,a8da	Directory-Größe, jedes 1-Bit ⇔ 1 Block
a8db,a8dc	Anzahl zu prüfender Extents für Disc-Change-Erkennung
a8dd,a8de	Anzahl belegter Systemspuren
a8df	Nummer des ersten Sektors einer Spur
a8e0	Sektoren pro Spur
a8e1	Länge der Gap3 beim Sektor-Lesen-Schreiben
a8e2	Länge der Gap3 beim Formatieren
a8e3	Füll-Byte beim Formatieren
a8e4	Sektorlänge (in der FDC-Codierung)
a8e5	Records pro Sektor
a8e6	aktuelle Spurnummer
a8e7	Flag für Recalibrate vor Spursuchen
a8e8	Flag für Login vor jedem Diskettenzugriff
a8e9-a8f8	Puffer für Checksummen
a8f9-a90f	23 Bytes für die Block-Belegungstabelle
a910-a91f	Disk Parameter Header für Laufwerk A:
a910,a911	Umsetzung des Skew-Faktors (unbenutzt)

CPC 464/664/6128	Bedeutung der Speicherstelle(n)
a912,a913	aktuelle Spur
a914,a915	aktueller Sektor
a916,a917	aktuelle Directory-Nummer
a918,a919	Zeiger auf den Directory-I/O-Puffer
a91a,a91b	Zeiger auf Disk Parameter Block (Laufwerk A)
a91c,a91d	Zeiger auf Puffer für Checksummen
a91e,a91f	Zeiger auf Block-Belegungstabelle
a920-a92f	Disk Parameter Header für Laufwerk B:
a920,a921	Umsetzung des Skew-Faktors (unbenutzt)
a922,a923	aktuelle Spur
a924,a925	aktueller Sektor
a926,a927	aktuelle Directory-Nummer
a928,a929	Zeiger auf den Directory-I/O-Puffer
a92a,a92b	Zeiger auf Disk Parameter Block (Laufwerk B)
a92c,a92d	Zeiger auf Puffer für Checksummen
a92e,a92f	Zeiger auf Block-Belegungstabelle
a930-a9af	Puffer für einen Directory-Record
a9b0-abaf	Puffer für einen Sektor

Anhang C

Der Z80

Die Befehle des Z80 sortiert nach ihrem Opcode

(In der Tabelle sind auch ein paar Illegals enthalten, also Operationen, die nicht zur Z80-Spezifikation gehören, aber von jedem normalen Z80 ausgeführt werden. Diese kann man meist mit Assemblerprogrammen nicht direkt eingeben, statt dessen muß man mit DEFB o.ä. arbeiten. Die Illegals sind mit * gekennzeichnet.)

Opcode		Befehls-Mnemonik		
dez	hex	sofort	nach CBhex	nach EDhex
0	00	nop	rlc b	
1	01	ld bc,NN	rlc c	
2	02	ld (bc),a	rlc d	
3	03	inc bc	rlc e	
4	04	inc b	rlc h	
5	05	dec b	rlc l	
6	06	ld b,N	rlc (hl)	
7	07	rlca	rlc a	
8	08	ex af,af'	rrc b	
9	09	add hl,bc	rrc c	
10	0a	ld a,(bc)	rrc d	
11	0b	dec bc	rrc e	
12	0c	inc c	rrc h	
13	0d	dec c	rrc l	
14	0e	ld c,N	rrc (hl)	
15	0f	rrca	rrc a	
16	10	djnz DIS	rl b	
17	11	ld de,NN	rl c	
18	12	ld (de),a	rl d	
19	13	inc de	rl e	
20	14	inc d	rl h	
21	15	dec d	rl l	
22	16	ld d,N	rl (hl)	
23	17	rla	rl a	
24	18	jr DIS	rr b	

Opcode dez	hex	Befehls-Mnemonik sofort	nach CBhex	nach EDhex
25	19	add hl,de	rr c	
26	1a	ld a,(de)	rr d	
27	1b	dec de	rr e	
28	1c	inc e	rr h	
29	1d	dec e	rr l	
30	1e	ld e,N	rr (hl)	
31	1f	rra	rr a	
32	20	jr nz,DIS	sla b	
33	21	ld hl,NN	sla c	
34	22	ld (NN),hl	sla d	
35	23	inc hl	sla e	
36	24	inc h	sla h	
37	25	dec h	sla l	
38	26	ld h,N	sla (hl)	
39	27	daa	sla a	
40	28	jr z,DIS	sra b	
41	29	add hl,hl	sra c	
42	2a	ld hl,(NN)	sra d	
43	2b	dec hl	sra e	
44	2c	inc l	sra h	
45	2d	dec l	sra l	
46	2e	ld l,N	sra (hl)	
47	2f	cpl	sra a	
48	30	jr nc,DIS	sll b *	
49	31	ld sp,NN	sll c *	
50	32	ld (NN),a	sll d *	
51	33	inc sp	sll e *	
52	34	inc (hl)	sll h *	
53	35	dec (hl)	sll l *	
54	36	ld (hl),N	sll (hl) *	
55	37	scf	sll a *	
56	38	jr c,DIS	srl b	
57	39	add hl,sp	srl c	
58	3a	ld a,(NN)	srl d	
59	3b	dec sp	srl e	
60	3c	inc a	srl h	
61	3d	dec a	srl l	
62	3e	ld a,N	srl (hl)	
63	3f	ccf	srl a	
64	40	ld b,b	bit 0,b	in b,(c)
65	41	ld b,c	bit 0,c	out (c),b
66	42	ld b,d	bit 0,d	sbc hl,bc
67	43	ld b,e	bit 0,e	ld (NN),bc
68	44	ld b,h	bit 0,h	neg
69	45	ld b,l	bit 0,l	retn

Opcode dez	hex	Befehls-Mnemonik sofort	nach CBhex	nach EDhex
70	46	ld b,(hl)	bit 0,(hl)	im 0
71	47	ld b,a	bit 0,a	ld i,a
72	48	ld c,b	bit 1,b	in c,(c)
73	49	ld c,c	bit 1,c	out (c),c
74	4a	ld c,d	bit 1,d	adc hl,bc
75	4b	ld c,e	bit 1,e	ld bc,(NN)
76	4c	ld c,h	bit 1,h	
77	4d	ld c,l	bit 1,l	reti
78	4e	ld c,(hl)	bit 1,(hl)	
79	4f	ld c,a	bit 1,a	ld r,a
80	50	ld d,b	bit 2,b	in d,(c)
81	51	ld d,c	bit 2,c	out (c),d
82	52	ld d,d	bit 2,d	sbc hl,de
83	53	ld d,e	bit 2,e	ld (NN),de
84	54	ld d,h	bit 2,h	
85	55	ld d,l	bit 2,l	
86	56	ld d,(hl)	bit 2,(hl)	im 1
87	57	ld d,a	bit 2,a	ld a,i
88	58	ld e,b	bit 3,b	in e,(c)
89	59	ld e,c	bit 3,c	out (c),e
90	5a	ld e,d	bit 3,d	adc hl,de
91	5b	ld e,e	bit 3,e	ld de,(NN)
92	5c	ld e,h	bit 3,h	
93	5d	ld e,l	bit 3,l	
94	5e	ld e,(hl)	bit 3,(hl)	im 2
95	5f	ld e,a	bit 3,a	ld a,r
96	60	ld h,b	bit 4,b	in h,(c)
97	61	ld h,c	bit 4,c	out (c),h
98	62	ld h,d	bit 4,d	sbc hl,hl
99	63	ld h,e	bit 4,e	ld (NN),hl
100	64	ld h,h	bit 4,h	
101	65	ld h,l	bit 4,l	
102	66	ld h,(hl)	bit 4,(hl)	
103	67	ld h,a	bit 4,a	rrd
104	68	ld l,b	bit 5,b	in l,(c)
105	69	ld l,c	bit 5,c	out (c),l
106	6a	ld l,d	bit 5,d	adc hl,hl
107	6b	ld l,e	bit 5,e	ld hl,(NN)
108	6c	ld l,h	bit 5,h	
109	6d	ld l,l	bit 5,l	
110	6e	ld l,(hl)	bit 5,(hl)	
111	6f	ld l,a	bit 5,a	rld
112	70	ld (hl),b	bit 6,b	in f,(c)
113	71	ld (hl),c	bit 6,c	

Opcode dez	hex	Befehls-Mnemonik sofort	nach CBhex	nach EDhex
114	72	ld (hl),d	bit 6,d	sbc hl,sp
115	73	ld (hl),e	bit 6,e	ld (NN),sp
116	74	ld (hl),h	bit 6,h	
117	75	ld (hl),l	bit 6,l	
118	76	halt	bit 6,(hl)	
119	77	ld (hl),a	bit 6,a	
120	78	ld a,b	bit 7,b	in a,(c)
121	79	ld a,c	bit 7,c	out (c),a
122	7a	ld a,d	bit 7,d	adc hl,sp
123	7b	ld a,e	bit 7,e	ld sp,(NN)
124	7c	ld a,h	bit 7,h	
125	7d	ld a,l	bit 7,l	
126	7e	ld a,(hl)	bit 7,(hl)	
127	7f	ld a,a	bit 7,a	
128	80	add a,b	res 0,b	
129	81	add a,c	res 0,c	
130	82	add a,d	res 0,d	
131	83	add a,e	res 0,e	
132	84	add a,h	res 0,h	
133	85	add a,l	res 0,l	
134	86	add a,(hl)	res 0,(hl)	
135	87	add a,a	res 0,a	
136	88	adc a,b	res 1,b	
137	89	adc a,c	res 1,c	
138	8a	adc a,d	res 1,d	
139	8b	adc a,e	res 1,e	
140	8c	adc a,h	res 1,h	
141	8d	adc a,l	res 1,l	
142	8e	adc a,(hl)	res 1,(hl)	
143	8f	adc a,a	res 1,a	
144	90	sub b	res 2,b	
145	91	sub c	res 2,c	
146	92	sub d	res 2,d	
147	93	sub e	res 2,e	
148	94	sub h	res 2,h	
149	95	sub l	res 2,l	
150	96	sub (hl)	res 2,(hl)	
151	97	sub a	res 2,a	
152	98	sbc a,b	res 3,b	
153	99	sbc a,c	res 3,c	
154	9a	sbc a,d	res 3,d	
155	9b	sbc a,e	res 3,e	
156	9c	sbc a,h	res 3,h	
157	9d	sbc a,l	res 3,l	
158	9e	sbc a,(hl)	res 3,(hl)	

Opcode dez	hex	Befehls-Mnemonik sofort	nach CBhex	nach EDhex
159	9f	sbc a,a	res 3,a	
160	a0	and b	res 4,b	ldi
161	a1	and c	res 4,c	cpi
162	a2	and d	res 4,d	ini
163	a3	and e	res 4,e	outi
164	a4	and h	res 4,h	
165	a5	and l	res 4,l	
166	a6	and (hl)	res 4,(hl)	
167	a7	and a	res 4,a	
168	a8	xor b	res 5,b	ldd
169	a9	xor c	res 5,c	cpd
170	aa	xor d	res 5,d	ind
171	ab	xor e	res 5,e	outd
172	ac	xor h	res 5,h	
173	ad	xor l	res 5,l	
174	ae	xor (hl)	res 5,(hl)	
175	af	xor a	res 5,a	
176	b0	or b	res 6,b	ldir
177	b1	or c	res 6,c	cpir
178	b2	or d	res 6,d	inir
179	b3	or e	res 6,e	otir
180	b4	or h	res 6,h	
181	b5	or l	res 6,l	
182	b6	or (hl)	res 6,(hl)	
183	b7	or a	res 6,a	
184	b8	cp b	res 7,b	lddr
185	b9	cp c	res 7,c	cpdr
186	ba	cp d	res 7,d	indr
187	bb	cp e	res 7,e	otdr
188	bc	cp h	res 7,h	
189	bd	cp l	res 7,l	
190	be	cp (hl)	res 7,(hl)	
191	bf	cp a	res 7,a	
192	c0	ret nz	set 0,b	
193	c1	pop bc	set 0,c	
194	c2	jp nz,NN	set 0,d	
195	c3	jp NN	set 0,e	
196	c4	call nz,NN	set 0,h	
197	c5	push bc	set 0,l	
198	c6	add a,N	set 0,(hl)	
199	c7	rst 0	set 0,a	
200	c8	ret z	set 1,b	
201	c9	ret	set 1,c	
202	ca	jp z,NN	set 1,d	

Opcode dez	hex	Befehls-Mnemonik sofort	nach CBhex	nach EDhex
203	cb	* prefix *	set 1,e	
204	cc	call z,NN	set 1,h	
205	cd	call NN	set 1,l	
206	ce	adc a,NN	set 1,(hl)	
207	cf	rst 8	set 1,a	
208	d0	ret nc	set 2,b	
209	d1	pop de	set 2,c	
210	d2	jp nc,NN	set 2,d	
211	d3	out (N),a	set 2,e	
212	d4	call nc,NN	set 2,h	
213	d5	push de	set 2,l	
214	d6	sub N	set 2,(hl)	
215	d7	rst 16	set 2,a	
216	d8	ret c	set 3,b	
217	d9	exx	set 3,c	
218	da	jp c,NN	set 3,d	
219	db	in a,(N)	set 3,e	
220	dc	call c,NN	set 3,h	
221	dd	* prefix ix *	set 3,l	
222	de	sbc a,N	set 3,(hl)	
223	df	rst 24	set 3,a	
224	e0	ret po	set 4,b	
225	e1	pop hl	set 4,c	
226	e2	jp po,NN	set 4,d	
227	e3	ex (sp),hl	set 4,e	
228	e4	call po,NN	set 4,h	
229	e5	push hl	set 4,l	
230	e6	and N	set 4,(hl)	
231	e7	rst 32	set 4,a	
232	e8	ret pe	set 5,b	
233	e9	jp (hl)	set 5,c	
234	ea	jp pe,NN	set 5,d	
235	eb	ex de,hl	set 5,e	
236	ec	call pe,NN	set 5,h	
237	ed	* prefix *	set 5,l	
238	ee	xor N	set 5,(hl)	
239	ef	rst 40	set 5,a	
240	f0	ret p	set 6,b	
241	f1	pop af	set 6,c	
242	f2	jp p,NN	set 6,d	
243	f3	di	set 6,e	
244	f4	call p,NN	set 6,h	
245	f5	push af	set 6,l	
246	f6	or N	set 6,(hl)	
247	f7	rst 48	set 6,a	

Opcode dez	hex	Befehls-Mnemonik sofort	nach CBhex	nach EDhex
248	f8	ret m	set 7,b	
249	f9	ld sp,hl	set 7,c	
250	fa	jp m,NN	set 7,d	
251	fb	ei	set 7,e	
252	fc	call m,NN	set 7,h	
253	fd	* prefix iy *	set 7,l	
254	fe	cp N	set 7,(hl)	
255	ff	rst 56	set 7,a	

Befehlssatz des Z80, sortiert nach Funktionen

BENUTZTE ABKÜRZUNGEN

dis	=	Adreßdistanz:		–128 .... +127
n	=	Byte-Konstante:	0 .... +255 oder	–128 .... +127
nn	=	Word-Konstante:	0 .. +65535 oder	–32768 .. +32767
r	=	Register:	A,B,C,D,E,H,L	
s	=	Byte-Quelle:	A,B,C,D,E,H,L,(HL),(IX+dis),(IY+dis),n	
d	=	Byte-Ziel:	A,B,C,D,E,H,L,(HL),(IX+dis),(IY+dis)	
b	=	Bit-Nummer:	0 ... 7	

8-BIT-LADEBEFEHLE

LD	r,s	
LD	d,r	
LD	d,n	
LD	A,q	q=s,(BC),(DE),(nn),I,R
LD	z,A	z=d,(BC),(DE),(nn),I,R

16-BIT-LADEBEFEHLE

LD	rr,(nn)	rr=	BC,DE, HL,IX,IY,SP
LD	(nn),rr	rr=	BC,DE, HL,IX,IY,SP
LD	SP,rr	rr=	HL,IX,IY
PUSH	rr	rr=	AF, BC,DE, HL,IX,IY
POP	rr	rr=	AF, BC,DE, HL,IX,IY

16-BIT-REGISTER-AUSTAUSCH

EX	AF,A'F'	
EXX		BC,DE,HL
EX	DE,HL	
EX	(SP),rr	rr=HL,IX,IY

BLOCK-TRANSPORTBEFEHLE

LDI	→	LD (DE),(HL) : INC DE : INC HL : DEC BC
LDIR	→	LDI bis BC=0
LDD	→	LD (DE),(HL) : DEC DE : DEC HL : DEC BC
LDDR	→	LDD bis BC=0

BLOCK-SUCHBEFEHLE

CPI	→	CP (HL) : INC HL : DEC BC
CPIR	→	CPI bis A=(HL) oder BC=0
CPD	→	CP (HL) : DEC HL : DEC BC
CPDR	→	CPD bis A=(HL) oder BC=0

BLOCK-I/O-BEFEHLE

Diese sind beim Schneider CPC nicht anwendbar!

8-BIT ARITHMETISCHE UND LOGISCHE OPERATIONEN

SUB	s	AND	s	INC	d
SBC	s	OR	s	DEC	d
ADD	s	XOR	s		
ADC	s	CP	s		

CPL → LD A,255-A
NEG → LD A, -A

16-BIT ARITHMETISCHE OPERATIONEN

ADD dd,rr	rr=BC,DE,dd,SP	dd=HL,IX,IY
ADC HL,rr	rr=BC,DE,HL,SP	

SBC HL,rr	rr=BC,DE,HL,SP
INC dd	dd=BC,DE,HL,IX,IY,SP
DEC dd	dd=BC,DE,HL,IX,IY,SP

8-BIT BIT-SCHIEBEBEFEHLE

RL	s	Ringshift mit CY als 9. Bit links
RR	s	Ringshift mit CY als 9. Bit rechts
RLC	s	Ringshift links
RRC	s	Ringshift rechts
SLA	s	Shift links und 0 → Bit 0
SRA	s	Shift rechts und Bit 7 belassen
SLL	s	Shift links und 1 → Bit 0 (Illegal)
SRL	s	Shift rechts und 0 → Bit 7

8-BIT NIBBLE-SCHIEBEBEFEHLE

RLD	→	A0123 → (HL)0123 → (HL)4567 → A0123
RRD	→	A0123 ← (HL)0123 ← (HL)4567 ← A0123

BIT-BEFEHLE

BIT	b,d	Teste Bit b in Byte d
SET	b,d	Setze Bit auf 1
RES	b,d	Setze Bit auf 0

EIN- UND AUSGABEBEFEHLE

IN	A,(n)	
IN	r,(C)	de facto: IN r,(BC)
OUT	(n),A	
OUT	(C),r	OUT (BC),r

Block-Ein- und -Ausgabe-Befehle sind beim Schneider CPC nicht anwendbar!

SPRUNGBEFEHLE

JP	nn
JR	dis

JP	(rr)	rr=HL,IX,IY
JP	c,nn	c=C,NC,Z,NZ,M,P,PO,PE
JR	c,dis	c=C,NC,Z,NZ
DJNZ	dis	DEC B : JR NZ,dis

UNTERPROGRAMMBEFEHLE

CALL	nn	
CALL	c,nn	c=C,NC,Z,NZ,M,P,PO,PE
RST	v	v=0,8,16,24,32,40,48,56
RET		
RET	c	c=C,NC,Z,NZ,M,P,PO,PE
RETI		Return vom Interrupt
RETN		Return vom NMI; dieser ist beim CPC nicht so einfach zu benutzen!

SONSTIGE BEFEHLE

DAA	BCD-Korrektur einer Addition oder Subtraktion
SCF	C-Flag setzen
CCF	C-Flag umdrehen
EI	
DI	
IM0	Der Schneider CPC wird im Interrupt-Modus 1
IM1	betrieben!
IM2	
HALT	Warte auf Interrupt
NOP	

Wirkung der Z80-Befehle auf die Flags

Das Flag-Byte:

S	Z	–	H	–	P/V	N	C

Flag		Name	Bedeutung, wenn 0	Bedeutung, wenn 1
Bit	7	Signum	M/Minus/Negativ	P/Plus/Positiv
Bit	6	Zero	NZ/ungleich Null	Z/Zero/gleich Null
Bit	5	------		
Bit	4	Hilfs-Carry	--- wird vom Befehl DAA ausgewertet ---	
Bit	3	------		

Flag		Name	Bedeutung, wenn 0	Bedeutung, wenn 1
Bit	2	Parity	PO/ungerade Quersumme	PE/gerade Quersumme
		Overflow	PO/kein Überlauf	PE/Überlauf ins Vorzeichen-Bit
Bit	1	Add/Sub	--- wird vom Befehl DAA ausgewertet ---	
Bit	0	Carry	NC/kein Übertrag	C/Übertrag vom höchsten Bit

BENUTZTE ABKÜRZUNGEN

* – Flag wird entsprechend dem Ausgang der Operation gesetzt
0 – Flag ist nach der Operation immer zurückgesetzt
1 – Flag ist nach der Operation immer gesetzt
P – P/V-Flag wird im Sinne von Parity gesetzt
V – P/V-Flag wird im Sinne von Overflow gesetzt
? – Flag wird evtl. geändert, zeigt aber nichts an

Befehle	C	Z	P/V	S	Anmerkung
ADD A,s / ADC A,s	*	*	V	*	
SUB s / SBC A,s / CP s / NEG	*	*	V	*	
AND s / OR s / XOR s	0	*	P	*	
INC s / DEC s		*	V	*	
ADD rr,rr	*				
ADC rr,rr / SBC rr,rr	*	*	V	*	
RLA / RLCA /RRA /RRCA	*				1-Byte-Befehle für A
RL r / RLC r /RRr /RRCr	*	*	P	*	
SLA r / SRA r /SLLr /SRLr	*	*	P	*	
RLD / RRD		*	P	*	
DAA	*	*	P	*	
SCF / CCF	*				
IN r,(C)		*	P	*	
INI / IND /OUTI /OUTD		*	?	?	Block-I/O-Operationen sind
INIR / INDR /OTIR /OTDR		1	?	?	beim CPC nicht anwendbar!
LDI / LDD		?	*	?	PO, wenn BC=0, sonst PE
LDIR / LDDR /CPIR /CPDR		?	0	?	
CPI / CPD		*	*	?	Z wenn A=(HL) gefunden PO, wenn BC=0, sonst PE
LD A,I / LD A,R		*	IFF	*	IFF=Zustand des EI/DI-FlipFlop
BIT b,s		*	?	?	

Befehlscodes der Z80-Befehle nach Opcode-Gruppen

Es sind nur die Befehle aufgeführt, die sich von ihrem Code her in Gruppen zusammenfassen lassen. Die Codes der fehlenden Befehle können aus der vorhergehenden Tabelle entnommen werden. Die Opcodes werden nur mit den Bytes angegeben, die den Befehl klassifizieren. Das oder die Bytes für eventuell nötige Operanden werden nicht explizit angegeben.

Operationen, die das IX- und IY-Register anstelle von HL benutzen, sind auch nicht aufgeführt, da sie sich von den HL-Befehlen nur durch den Präfix #DD bzw. #FD unterscheiden. Doppelregisterbefehle wie LD HL,nnnn bekommen einfach den Präfix davorgesetzt:

```
DEFB   #DD          ' Praefix IX
DEFB   #21          ' Opcode: LD HL,nnnn
DEFW   adresse
```

Byte-Indirekt-HL-Operationen arbeiten dann indirekt zu IX bzw. IY mit einem Distanz-Byte, beispielsweise LD B,(IX+dis):

```
DEFB   #DD          ' Praefix IX
DEFB   #46          ' Opcode: LD B,(HL)
DEFB   dis          ' Distanz (Offset)
```

oder: `LD (IX+dis),zahl`

```
DEFB   #DD          ' Praefix IX
DEFB   #36          ' Opcode: LD (HL),nn
DEFB   dis          ' Distanz (Offset)
DEFB   zahl         ' Argument
```

Befehlsgruppe	Opcode (binär)	Bemerkung
Byte-Befehle		
LD r,r'	01.r..r'	r und r': 000→ B
LD r,n	00.r.110	001→ C
ADD A,r	10000.r.	010→ D
ADC A,r	10001.r.	011→ E
SUB r	10010.r.	100→ H
SBC A,r	10011.r.	101→ L
AND r	10100.r.	110→ (HL)
XOR r	10101.r.	111→ A
OR r	10110.r.	
CP r	10111.r.	Bit b: 000→ 0
INC r	00.r.100	001→ 1
DEC r	00.r.101	010→ 2
RLC r	11001011 00000.r.	011→ 3
RRC r	11001011 00001.r.	100→ 4
RL r	11001011 00010.r.	101→ 5

Befehlsgruppe	Opcode (binär)	Bemerkung
Byte-Befehle		
RR r	11001011 00011.r.	110→ 6
SLA r	11001011 00100.r.	111→ 7
SRA r	11001011 00101.r.	
SLL r	11001011 00110.r.	
SRL r	11001011 00111.r.	
BIT b,r	11001011 01.b..r.	
RES b,r	11001011 10.b..r.	
SET b,r	11001011 11.b..r.	
IN r,(c)	11101101 01.r.000	
OUT (c),r	11101101 01.r.001	
2-Byte-Befehle		
LD rr,nn	00rr0001	rr: 00→ BC
LD rr,(nn)	11101101 01rr1011	01→ DE
LD (nn),rr	11101101 01rr0011	10→ HL
ADD HL,rr	00rr1001	11→ SP
ADC HL,rr	11101101 01rr1010	
SBC HL,rr	11101101 01rr0010	
INC rr	00rr0011	
DEC rr	00rr1011	
PUSH rr	11rr0101	rr: 00 BC 01 DE
POP rr	11rr0001	10 HL 11 AF
Verzweigungen		
JP c,nn	11.c.01	
CALL c,NN	11.c.100	(Beding.) c:000 →NZ 001 →Z
RET c	11.c.000	010→ NC 011→ C
		100→ PO 101→ PE
		110→ P 111→M
JR cc,dis	001cc000	
	cc:	00 → NZ 01→ Z
		10 → NC 11→ C
RST n 8	11.n.111	0...n...7 → 0...n* 8...56

ILLEGALS

Außer SLL r gibt es noch weitere illegale Opcodes, die durch Kombinationen der verschiedenen Präfixe gebildet werden.

```
SLL  r          ; entspricht:          Register
                ;              CY <--- 76543210 <--- 1
```

Setzt man vor Byte-Operationen, die die Register H oder L benutzen, den Indexregister-Präfix #DD oder #FD, so wird das High- bzw. Low-Byte des entsprechenden Index-Registers benutzt:

```
DEFB #DD        ; entspricht:
LD   H,A        ; LD XH,A

DEFB #FD        ; entspricht:
LD   B,L        ; LD B,YL

DEFB #DD        ; entspricht:
LD   H,L        ; LD HX,LX
```

Bei Operationen und Funktionen funktioniert das entsprechend:

```
DEFB #FD        ; entspricht:
INC  H          ; INC HY

DEFB #DD        ; entspricht:
SBC  A,L        ; SBC A,LX
```

Ausführungszeiten für die einzelnen Befehle des Z80

Die Arbeitsgeschwindigkeit einer CPU hängt im allgemeinen von der Frequenz des Eingangstaktes ab. Da die Z80 im Schneider CPC aber nur mit jedem 4. Takt auf die Speicher zugreifen darf, ergibt sich als kleinstes Zeitraster genau eine Mikrosekunde (der Takt hat eine Frequenz von 4 MHz, also 4 Taktperioden pro Mikrosekunde).

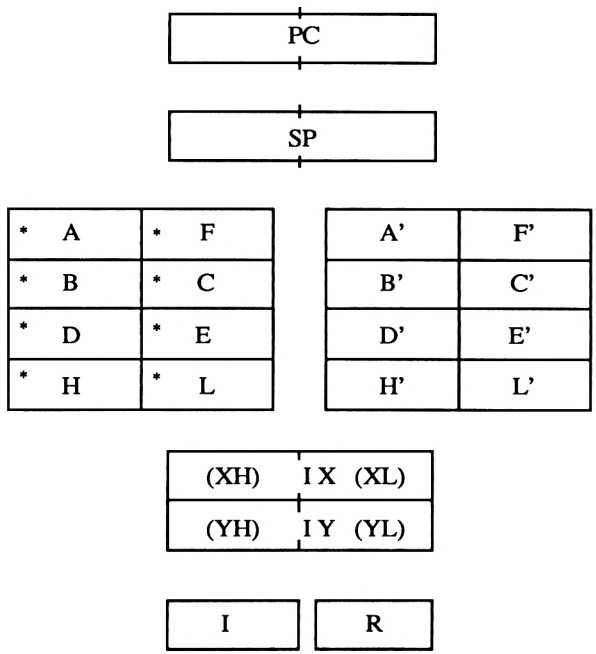
Die Befehls-Ausführungszeiten sind deshalb in Mikrosekunden angegeben. Diese Zeiten sind *nicht* auf andere Z80-Rechner übertragbar!

Befehlsgruppe		Zeit	Bemerkung
8-Bit-Ladebefehle			
LD r,r'		1	
LD r,n		2	r und r': A,B,C,D,E,H,L
LD r,(rr)	LD (rr),r	2	rr : AF,BC,DE,HL,SP
LD r,(IX+dis)	LD (IX+dis),r	5	n : Byte, direkt
LD (HL),n		3	nn : Word, direkt
LD (IX+dis),n		6	dis: Adreßdistanz, direkt
LD A,(nn)	LD (nn),A	4	
LD A,I LD A,R	LD I,A LD R,A	2	
16-Bit-Ladebefehle			
LD rr,nn		3	
LD IX,nn		4	
LD HL,(nn)	LD (nn),HL	5	
LD rr,(HL)	LD (nn),rr	6	
LD IX,(nn)	LD (nn),IX	6	
LD SP,HL		2	
LD SP,IX		3	
PUSH rr		4	
PUSH IX		5	
POP rr		3	
POP IX		4	
EXX EX AF,AF'	EX DE,HL	1	
EX (SP),HL		6	
EX (SP),IX		7	
Blockbefehle		i/ii	(i): nicht-letzter Durchgang (ii): letzter Durchgang
LDI	LDD	5	
CPI	CPD	4	
INI IND	OUTI OUTD	5	nicht sinnvoll anwendbar
LDIR	LDDR	6/5	
CPIR	CPDR	6/4	
OTIR OTDR	INIR INDR	-----	Block-I/O-Befehle nicht anwendbar!
Ein- und Ausgabebefehle			
IN A,(n)	OUT A,(n)	3	
IN r,(C)	OUT (C),r	4	

Befehlsgruppe		Zeit	Bemerkung
8-Bit arithmetische und logische Befehle			
oper. r		1	oper. = ADD, ADC
oper. n	oper. (HL)	2	SUB, SBC
oper. (IX+dis)		5	AND, XOR
INC r	DEC r	4	OR, CP
INC (HL)	DEC (HL)	3	
INC (IX+dis)	DEC (IX+dis)	6	
CPL		1	
NEG		2	
16-Bit-Arithmetik			
ADD HL,rr		3	
ADC HL,rr	SBC HL,rr	4	
ADD IX,rr		4	
INC rr	DEC rr	2	
INC IX	DEC IX	3	
8-Bit Schiebeoperationen			
RLCA RRCA RLA RRA		1	oper. = RLC, RRC
oper. r		2	RL, RR
oper. (HL)		4	SLA, SRA
oper. (IX+dis)		7	SLL, SRL
RLD RRD		5	
Bit Test- und Setzbefehle			
BIT b,r		2	b = Bitnummer
BIT b,(HL)		3	
BIT b,(IX+dis)		6	
SET b,r	RES b,r	2	
SET b,(HL)	RES b,(HL)	4	
SET b,(IX+dis)	RES (IX+dis)	7	
Sprünge und Unterprogramme		i/ii	
JP nn JP c,nn		3	c = Bedingung
JP (HL)		1	(i): wenn c nicht
JP (IX)		2	erfüllt ist und keine
JR dis		3	Verzweigung erfolgt

Befehlsgruppe	Zeit	Bemerkung
JR c, dis	2/3	(ii): wenn c erfüllt ist und das Programm verzweigt
DJNZ dis	3/4	
CALL nn	5	
CALL c, nn	3/5	
RST n	4	
RET	3	
RET c	2/4	
RETI RETN	4	
Verschiedenes:		
DAA CCF SCF	1	
EI DI NOP HALT	1	
IM 0 IM 1 IM 2	1	

DIE REGISTER DES Z80

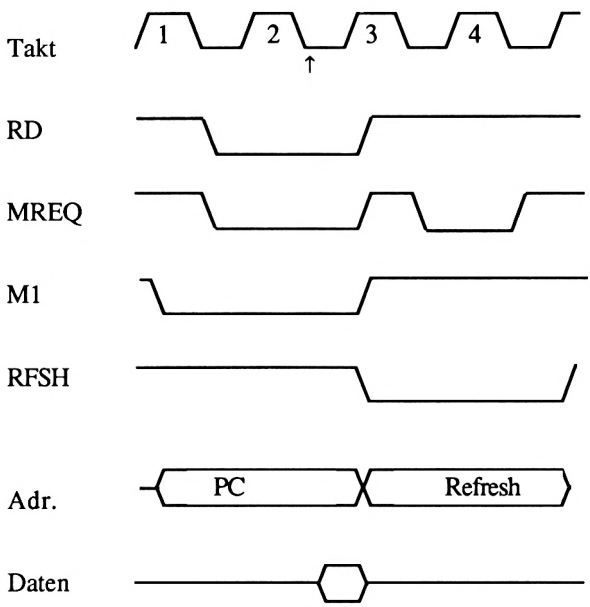


Die Grafik zeigt einen Überblick über die Register des Z80. Jedes Kästchen symbolisiert dabei normalerweise ein 8 Bit breites Register. Die Register, die auch bei dem Vorgänger des Z80, der 8080-CPU vorhanden waren, sind mit einem Stern * gekennzeichnet:

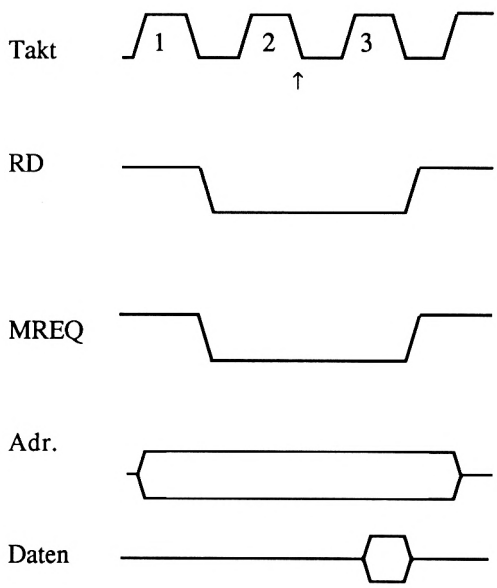
TIMING-DIAGRAMME FÜR DIE SPEICHER UND PERIPHERIEZUGRIFFE DES Z80

Die Taktflanken, an denen der Z80 den WAIT-Eingang testet, sind mit einem Pfeil ↑ markiert.

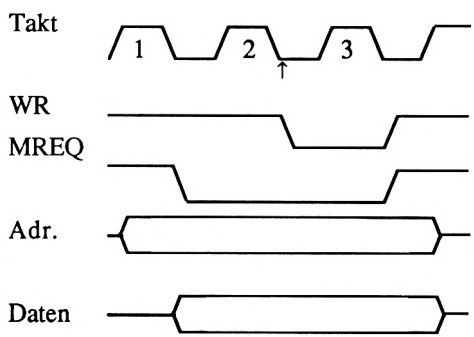
Instruction Opcode Fetch - M1-Cycle



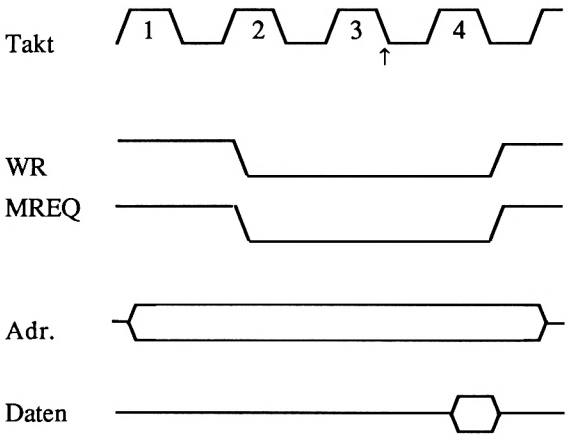
Memory Read Cycle



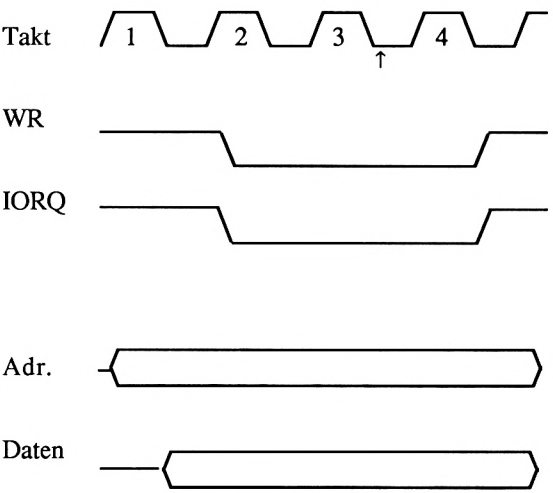
Memory Write Cycle



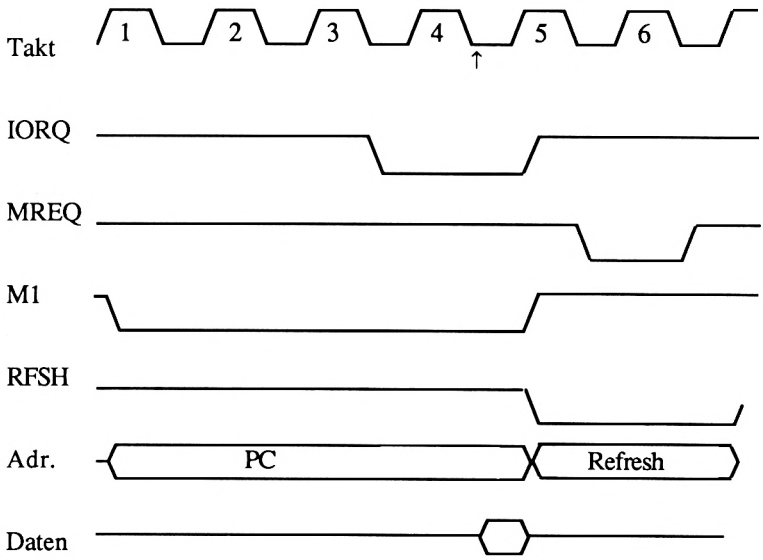
Input Cycle



Output Cycle



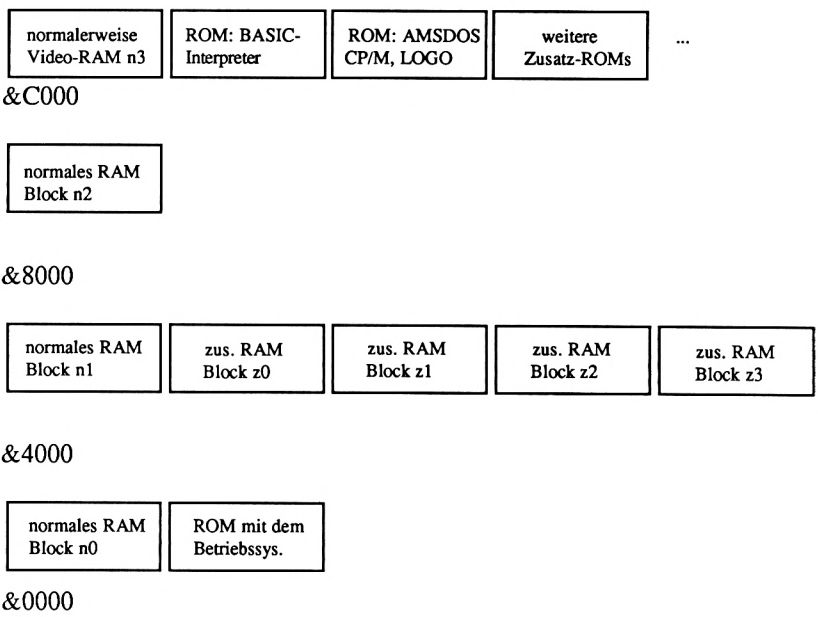
Interrupt Acknowledge Cycle



Anhang D

Speicher und Peripherie

Die Speicheraufteilung im CPC



Reservierte und frei verfügbare I/O-Adressen

angesprochener Baustein	Adresse: (Binär) FEDCBA98 76543210		Bemerkung
Gate Array, PAL	011111??	????????	
CRTC Video-Controller	10111100	????????	Register adressieren
	10111101	????????	Register beschreiben
	10111110	????????	reserviert
	10111111	????????	Register lesen
ROM select	110111??	????????	
Drucker-Port	111011??	????????	
8255 PIO I/O-Schnittstelle	11110100	????????	Port A Daten
	11110101	????????	Port B Daten
	11110110	????????	Port C Daten
	11110111	????????	Control-Register
Systembus Peripheriegeräte	111110??	011111??	Diskettenstation
	111110??	111110??	reserviert
	111110??	110111??	serielle Schnittstelle
	111110??	111011??	** frei verfügbar **
	111110??	111101??	** frei verfügbar **
	111110??	111110??	** frei verfügbar **

Die PIO 8255

Port-Adresse	Funktion
INP/OUT &F4xx →	Datenregister Port A:
	←→ Datenbus des PSG
INP &F5xx →	Datenregister Port B:
	Bit0 ← Vsync
	Bit 123 ← Firmenname
	Bit 4 ← 50/60Hz-Video
	Bit 5 ← EXP
	Bit 6 ← Drucker busy
	Bit 7 ← Data in vom Tape
OUT &F6xx →	Datenregister Port C:
	Bit 0–3 ← Tastaturzeile wählen
	Bit 4 ← Remote
	Bit 5 ← Data out zum Tape
	Bit 67 ← BC1 und BDIR zum PSG
OUT &F7xx →	Steuerregister

PIO-Steuerregister: Bit D7 gesetzt:

Bit	Funktion	wenn 0	wenn 1
0	Port C, Bits 0 bis 3	Ausgang	Eingang
1	Port B	Ausgang	Eingang
2	Gruppe B	Modus 0	Modus 1
3	Port C, Bits 4 bis 7	Ausgang	Eingang
4	Port A	Ausgang	Eingang
5	Gruppe A	Modus 0	Modus 1
6	Gruppe A	Modus 0 oder 1	Modus 2

PIO-Steuerregister: Bit D7 gelöscht:

Bit 0:	wird in das angewählte Bit von Port C kopiert: Bit 0 = 0 – löschen Bit 0 = 1 – setzen
Bits 1–3:	codieren binär die Nummer des Bits, das gesetzt oder gelöscht werden soll
Bits 4–6:	ohne Funktion

Die ULA

Port-Adresse: OUT &7FFF

Datenbyte	Bedeutung der einzelnen Bits im Datenbyte
&X0000iiii	Wählt das Farbregister für Tinte iiii an
&X0001????	Wählt das BORDER-Farbregister an
&X010nnnnn	Programmiert das gerade angewählte Farbregister mit der Farbe nnnnn (Paletten-Farbnummer)
&X100roumm	r=1 – Löscht den 52-Bildschirmzeilen-Zähler (Interrupt-Verzögerung) o=1 – Blendet oberes ROM aus u=1 – Blendet unteres ROM aus mm – Bestimmt Bildschirmmodus

Das PAL im CPC 6128

Port-Adresse: OUT &7FFF

Steuerung der RAM-Bankauswahl wie folgt:
(n)ormale RAM-Bank – (z)usätzliches RAM
0, 1, 2, 3: Block der jeweiligen Bank

Datenwort binär	CPU-Adreßviertel -0- -1- -2- -3-	Konfiguration wird verwendet bei:	normale Lage für Video-RAM
&X11000000	n0 – n1 – n2 – n3	Normalzustand	n3=Screen
&X11000001	n0 – n1 – n2 – z3	CP/M: BIOS, BDOS etc.	n1=Screen
&X11000010	z0 – z1 – z2 – z3	CP/M: TPA	(n1=Screen)
&X11000011	n0 – n3 – n2 – z3	CP/M: n3=Hush-Tabelle	(n1=Screen)
&X11000100	n0 – z0 – n2 – n3	Bankmanager	n3=Screen
&X11000101	n0 – z1 – n2 – n3	Bankmanager	n3=Screen
&X11000110	n0 – z2 – n2 – n3	Bankmanager	n3=Screen
&X11000111	n0 – z3 – n2 – n3	Bankmanager	n3=Screen

Das Eprom 27256

27256 – Arbeits-Modi				
CE	OE	A9	Vpp	D0-D7 / Modus
0	0	x	Vcc	Data out, normaler Lesezugriff
0	1	x	Vcc	hohe Impedanz, Output Disable
1	x	x	Vcc	hohe Impedanz, Stand By
0	0	12V	Vcc	Identifikationscode &04 (wenn A0=1)
0	1	x	Vpp	Data in, Programmieren
1	1	x	Vpp	hohe Impedanz, neutraler Zustand beim Programmieren
x	0	x	Vpp	Data out, Verify (Achtung bei NEC-Typen)

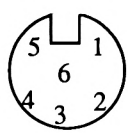
Das Eprom 27128

27128 – Arbeits-Modi

CE	OE	PGM	A9	Vpp	D0-D7 / Modus
0	0	1	x	Vcc	Data out, normaler Lesezugriff
0	1	1	x	Vcc	hohe Impedanz, Output Disable
1	x	x	x	Vcc	hohe Impedanz, Stand By
0	0	1	12V	Vcc	Identifikationscode &83 (wenn A0=1)
0	1	0	x	Vpp	Data in, Programmieren
0	0	1	x	Vpp	Data out, Verify
0	1	1	x	Vpp	hohe Impedanz, neutraler Zustand beim Programmieren
1	x	x	x	Vpp	hohe Impedanz, neutraler Zustand beim Programmieren

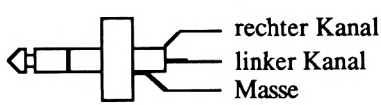
Die Anschlüsse am Schneider CPC

DER MONITORANSCHLUSS

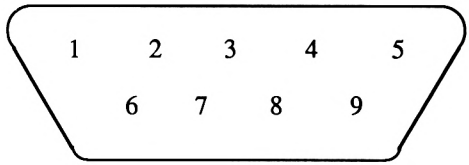


1 – R = Helligkeit des Rot-Anteils
2 – G = Helligkeit des Grün-Anteils
3 – B = Helligkeit des Blau-Anteils
4 – Synchronisation für den Strahl-
5 – rücklauf (horizontal & vertikal)
6 – Referenz = Masse-Anschluß 0 Volt
Mischsignal aus 1, 2, 3 und 4 = Luminanz (für eigene monochrome Monitore)

DER AUDIO-ANSCHLUSS

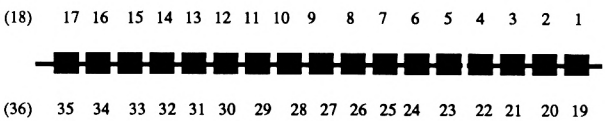


DER JOYSTICK-ANSCHLUSS



1 – Schalter-Ausgang für UP (nach oben)
2 – Schalter-Ausgang für DOWN (nach unten)
3 – Schalter-Ausgang für LEFT (nach links)
4 – Schalter-Ausgang für RIGHT (nach rechts)
5 – Schalter-Ausgang (nicht spezifiziert)
6 – Schalter-Ausgang für Feuer 2
7 – Schalter-Ausgang für Feuer 1
8 – Gemeinsamer Eingang für alle Schalter von Joystick 1 (COMMON 1)
9 – Gemeinsamer Eingang für alle Schalter von Joystick 2 (COMMON 2)

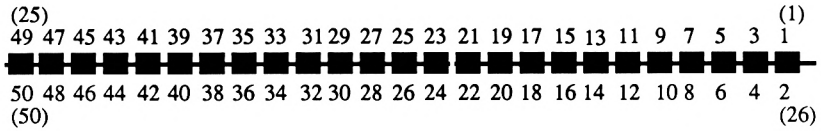
DER DRUCKER-PORT



1 ----- Strobe (0)
2 ----- Data Bit 0
3 ----- Data Bit 1
4 ----- Data Bit 2
5 ----- Data Bit 3
6 ----- Data Bit 4
7 ----- Data Bit 5
8 ----- Data Bit 6
9 ----- Data Bit 7 (GND)
11 <--- BUSY (1)

Alle anderen Leitungen sind mit Masse (GND) verbunden oder nicht angeschlossen.

DER EXPANSIONS-PORT (SYSTEMBUS)

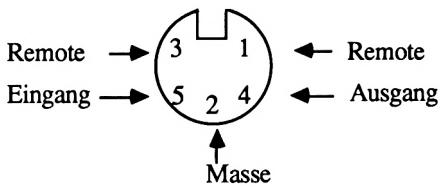


1 – Tonsignal
2 – GND

3 bis 18 – A15 bis A0
19 bis 26 – D7 bis D0

27 – Vcc
28 – (0) MREQ
29 – (0) M1
30 – (0) RFSH
31 – (0) IORQ
32 – (0) RD
33 – (0) WR
34 – (0) HALT
35 – (0) INT
36 – (0) NMI
37 – (0) BUSRQ
38 – (0) BUSAK
39 – (0) WAIT = (1) READY
40 – (0) BUS RESET
41 – (0) RESET
42 – (0) ROMEN
43 – (1) ROMDIS
44 – (0) RAMRD
45 – (1) RAMDIS
46 – (1) CURSOR
47 – (1) LIGHTPEN
48 – (0) EXPANSION
49 – GND = Masse
50 – Takt 4 MHz

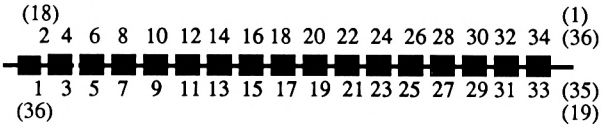
DER ANSCHLUSS FÜR DEN KASSETTENRECORDER AM CPC 664 UND 6128



DER RECORDERANSCHLUSS IM CPC 464

rot – +5 Volt Ausgang (für die Rechnerplatine, von Ein/Ausschalter)
schwarz – GND – Masse
weiß – +5 Volt Eingang (vom Monitor)
blau – Datenausgang (zum Rekorder)
grün – Dateneingang (vom Rekorder)
braun – GND – Masse
grau – Audio – Tonausgang zum Verstärker für den eingebauten Lautsprecher
gelb – Motorsteuerung

DER ANSCHLUSS FÜR DAS ZWEITE DISKETTENLAUFWERK

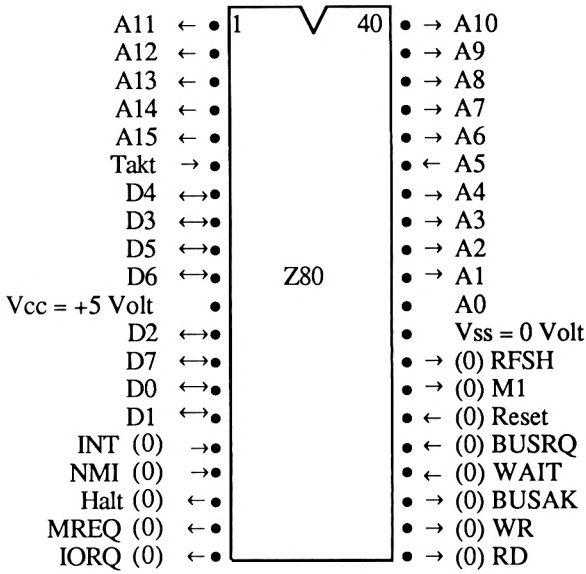


1 – (0) READY
3 – (0) SIDE 1 SELECT
5 – (0) READ DATA
7 – (0) WRITE PROTECT
9 – (0) TRACK 0
11 – (0) WRITE GATE
13 – (0) WRITE DATA
15 – (0) STEP
17 – (0) DIRECTION INWARDS
19 – (0) MOTOR ON
21 – n.c. (+5 Volt Drive A → Controller)
23 – (0) DRIVE B SELECT
25 – n.c. (DRIVE A SELECT)
27 – (0) INDEX
29 – n.c. (+5 Volt Drive A – Controller)
31 – n.c. (+5 Volt Drive A – Controller)
33 – n.c. (+5 Volt Drive A – Controller)

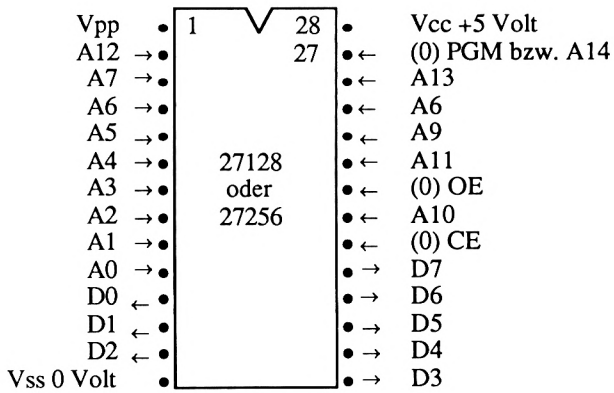
Alle anderen Leitungen sind mit Masse (GND) verbunden.

Die Anschlußbelegungen der wichtigsten ICs im CPC

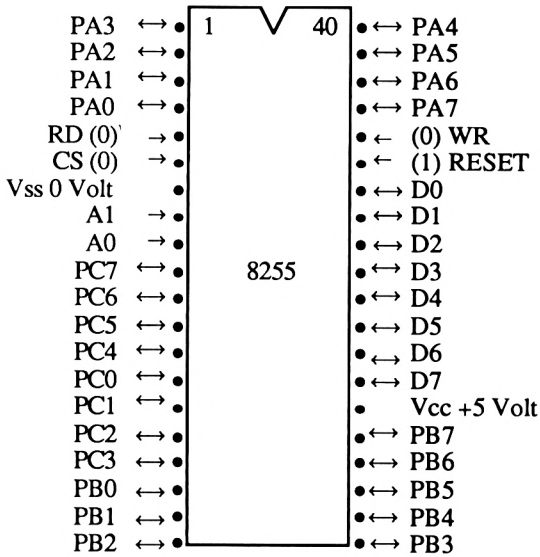
DIE CPU Z80



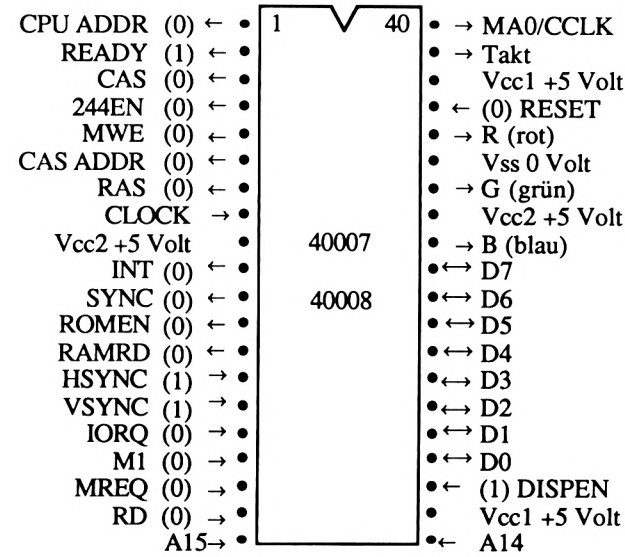
DIE EPROMs 27128 UND 27256



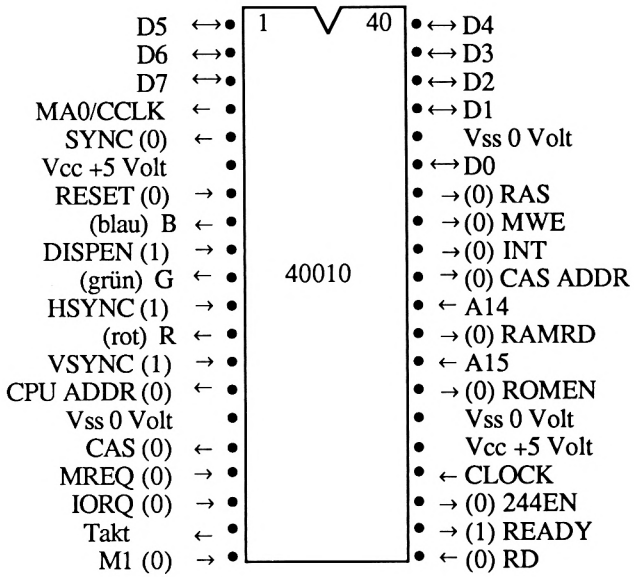
DIE PIO 8255



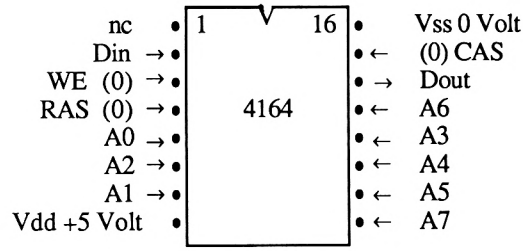
DIE ULA 40007 UND 40008 (CPC 464 UND 664)



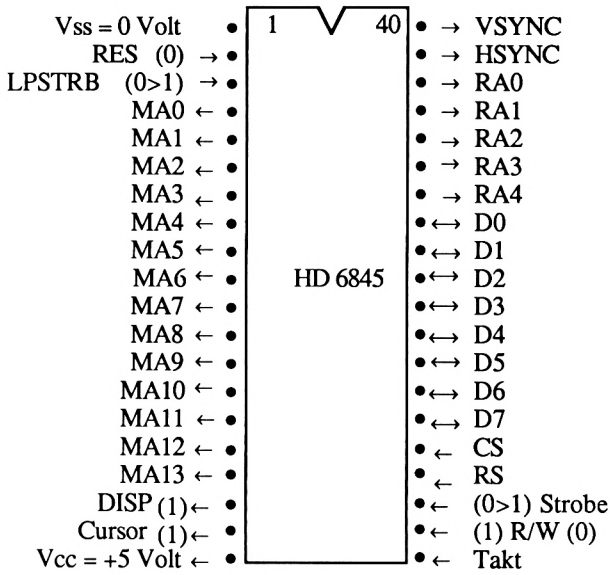
DIE ULA 40010 (CPC 6128)



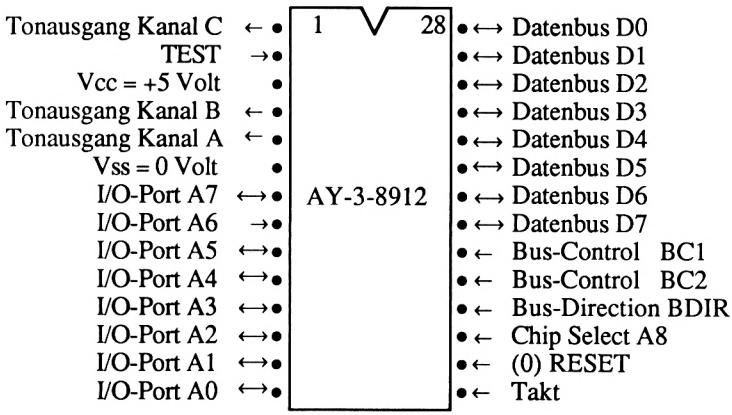
DAS RAM 4164



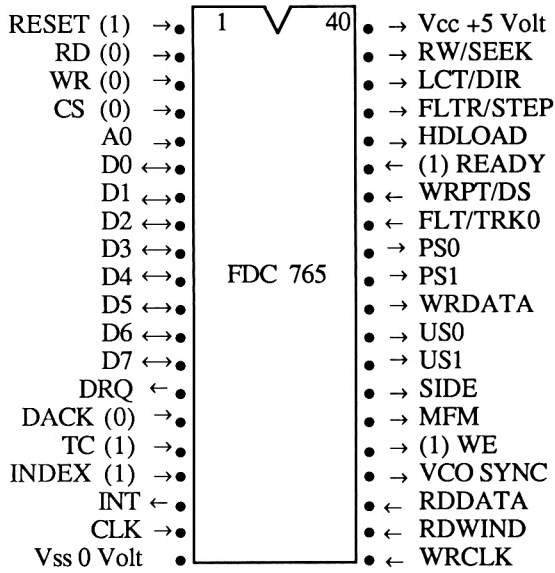
DER CRTC HD 6845



DER PSG AY-3-8912



DER FCD 765


RESET (1) →
RD (0) →
WR (0) →
CS (0) →
A0 →
D0 ↔
D1 ↔
D2 ↔
D3 ↔
D4 ↔
D5 ↔
D6 ↔
D7 ↔
DRQ ←
DACK (0) →
TC (1) →
INDEX (1) →
INT ←
CLK →
Vss 0 Volt
1
40
FDC 765
→ Vcc +5 Volt
→ RW/SEEK
→ LCT/DIR
→ FLTR/STEP
→ HDLOAD
← (1) READY
← WRPT/DS
← FLT/TRK0
→ PS0
→ PS1
→ WRDATA
→ US0
→ US1
→ SIDE
→ MFM
→ (1) WE
→ VCO SYNC
← RDDATA
← RDWIND
← WRCLK

Anhang E

Die Tastatur

Tastennummern

Die Tastatur des Schneider CPC ist an eine Drahtmatrix mit 8 * 10 Leitungen angeschlossen und wird vom Betriebssystem 50mal in der Sekunde überprüft. Die folgende Grafik zeigt die Lage der einzelnen Tasten in der Matrix und daraus resultierend ihre Tastennummern. Dabei werden mit den Klammern folgende Lagen symbolisiert:

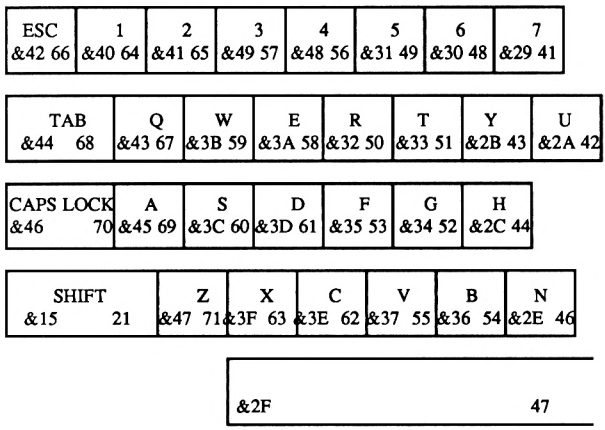
(...) → Zehnerblock oder Cursor-Taste
[...] → Joystick 0 (normaler Joystick)
{...} → Joystick 1 (zweiter Joystick)

Byte\Bit	0	1	2	3	4	5	6	7
0	(hoch)	(rechts)	(runter)	(9)	(6)	(3)	(ENTER)	(.)
8	(links)	(COPY)	(7)	(8)	(5)	(1)	(2)	(0)
16	CLR	[	ENTER	]	(4)	SHIFT	\	CTRL
24	^	–	@	P	;	:	/	.
32	0	9	O	I	L	K	M	,
40	8	7	U	Y	H	J	N	SPACE
48	6	5	R	T	G	F	B	V
! 48	{hoch}	{runter}	{links}	{rechts}	{Feuer1}	{Feuer2}	{n.c.}	
56	4	3	E	W	S	D	C	X
64	1	2	ESC	Q	TAB	A	CAPSLOCK	Z
72	[hoch]	[runter]	[links]	[rechts]	[Feuer1]	[Feuer2]	[n.c.]	DEL

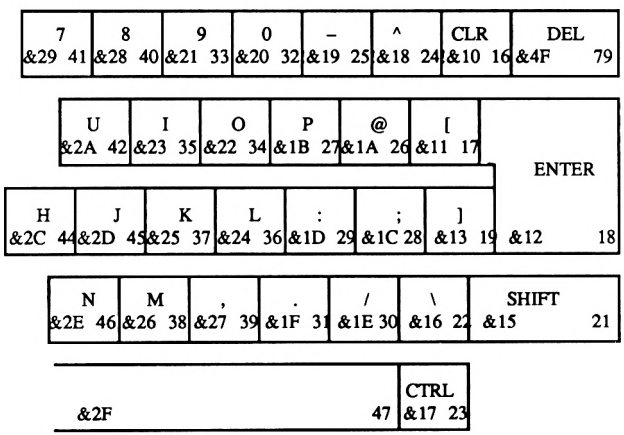
Schaubilder der Tastatur

Folgende Tabellen enthalten Schaubilder der Tastatur mit Tastenaufdruck und Tastennummer in Hexadezimal und Dezimal. Die Tastaturen des CPC 464 und 664 sind identisch, nur sind beim 664 die Cursortasten größer ausgefallen. Beim 6128 sind einige der Sondertasten verlegt worden, außerdem wurde der Cursorblock unten in den Zehnerblock integriert. Es sind aber bei allen drei Rechnertypen die gleichen Tasten vorhanden, und sie haben auch noch dieselben Codes.

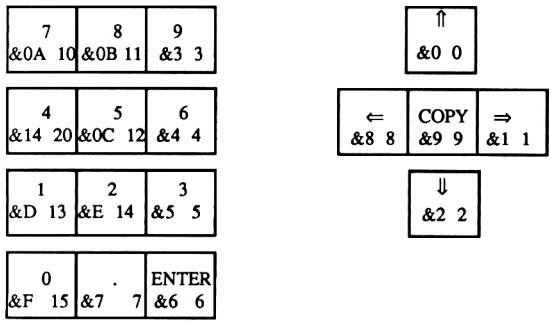
CPC 464/664: linke Hälfte des Haupt-Tastenfeldes



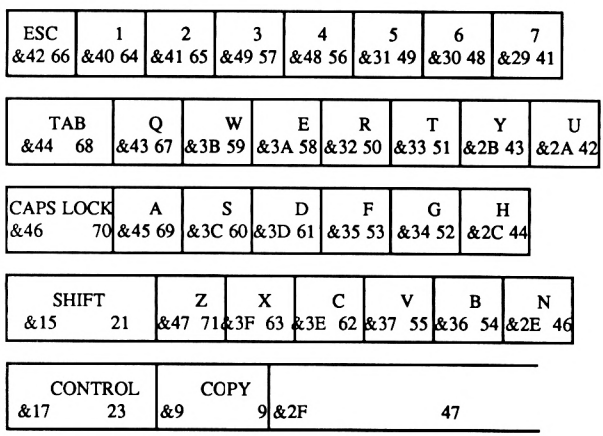
CPC 464/664: rechte Hälfte des Haupt-Tastenfeldes



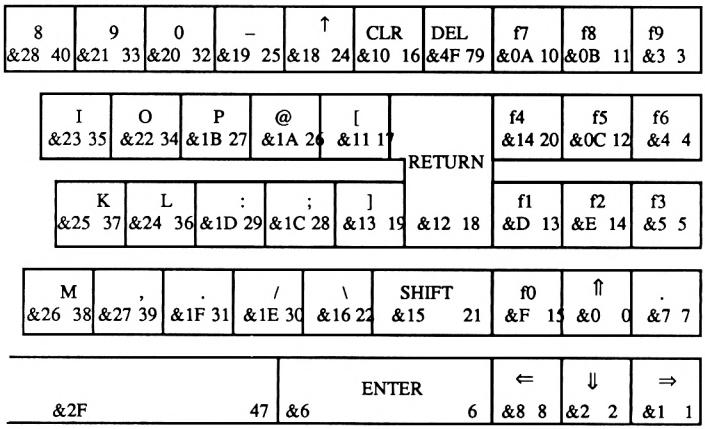
Zehnerblock CPC 464/664 Cursorblock



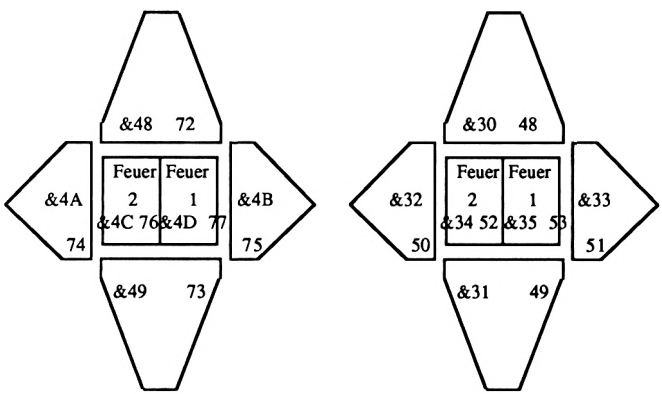
CPC 6128: linke Hälfte des Haupt-Tastenfeldes



CPC 6128: rechte Hälfte des Haupt-Tastenfeldes Cursor/Zehnerblock



erster Joystick 0 zweiter Joystick 1



Die Standard-Tastenübersetzung

Die Informationen sind wie folgt aufgebaut:

1. Zeile: Tastennummer (dezimal), aufgedrucktes Zeichen
   (*, wenn Auto-Repeat erlaubt)
3. Zeile: Tastenbelegung mit Control
4. Zeile: Belegung mit SHIFT
5. Zeile: Belegung Solo

0 ↑ *	1 → *	2 ↓ *	3 F9	4 F6
&F8 248	&FB 251	&F9 249	&89 137	&86 134
&F4 244	&F7 247	&F5 245	&89 137	&86 134
&F0 240	&F3 243	&F1 241	&89 137	&86 134
5 F3	6 ENTER	7 .	8 ← *	9 COPY *
&83 131	&8C 140	&8A 138	&FA 250	&E0 224
&83 131	&8B 139	&8A 138	&F6 246	&E0 224
&83 131	&8B 139	&8A 138	&F2 242	&E0 224
10 F7	11 F8	12 F5	13 F1	14 F2
&87 135	&88 136	&85 133	&81 129	&82 130
&87 135	&88 136	&85 133	&81 129	&82 130
&87 135	&88 136	&85 133	&81 129	&82 130
15 F0	16 CLR *	17 [*	18 ENTER	19] *
&80 128	&10 16	&1B 27	&0D 13	&1D 29
&80 128	&10 16	{ &7B 123	&0D 13	} &7D 125
&80 128	&10 16	[&5B 91	&0D 13	] &5D 93
20 F4	21 SHIFT	22 \ *	23 CTRL	24 ↑ *
&84 132	&FF 255	&1C 28	&FF 255	&1E 30
&84 132	&FF 255	‘ &60 96	&FF 255	&A3 163
&84 132	&FF 255	\ &5C 92	&FF 255	↑ &5E 94
25 – *	26 @ *	27 P *	28 ; *	29 : *
&FF 255	&00 0	&10 16	&FF 255	&FF 255
= &3D 61	&7C 124	P &50 80	+ &2B 43	* &2A 42
– &2D 45	@ &40 64	P &70 112	; &3B 59	: &3A 58

30 / *	31 . *	32 0 *	33 9 *	34 O *
&FF 255	&FF 255	&1F 31	&FF 255	&0F 15
? &3F 63	> &3E 62	– &5F 95	) &29 41	O &4F 79
/ &2F 47	. &2E 46	0 &30 48	9 &39 57	o &6F 111
35 I *	36 L *	37 K *	38 M *	39 , *
&09 9	&0C 12	&0B 11	&0D 13	&FF 255
I &49 73	L &4C 76	K &4B 75	M &4D 77	< &3C 60
i &69 105	l &6C 108	k &6B 107	m &6D 109	, &2C 44
40 8 *	41 7 *	42 U *	43 Y *	44 H *
&FF 255	&FF 255	&15 21	&19 25	&08 8
(&28 40	, &27 39	U &55 85	Y &59 89	H &48 72
8 &38 56	7 &37 55	u &75 117	y &79 121	h &68 104
45 J *	46 N *	47 *	48 6 *	49 5 *
&0A 10	&0E 14	&FF 255	&FF 255	&FF 255
J &4A 74	N &4E 78	&20 32	& &26 38	% &25 37
j &6A 106	n &6E 110	&20 32	6 &36 54	5 &35 53
50 R *	51 T *	52 G *	53 F *	54 B *
&12 18	&14 20	&07 7	&06 6	&02 2
R &52 82	T &54 84	G &47 71	F &46 70	B &42 66
r &72 114	t &74 116	g &67 103	f &66 102	b &62 98
55 V *	56 4 *	57 3 *	58 E *	59 W *
&16 22	&FF 255	&FF 255	&05 5	&17 23
V &56 86	$ &24 36	# &23 35	E &45 69	W &57 87
v &76 118	4 &34 52	3 &33 51	e &65 101	w &77 119
60 S *	61 D *	62 C *	63 X *	64 1 *
&13 19	&16 4	&03 3	&18 24	&FF 225
S &53 83	D &56 86	C &43 67	X &58 88	&21 33
s &73 115	d &76 100	c &63 99	x &78 120	1 &31 49
65 2 *	66 ESC	67 Q *	68 TAB	69 A *
~ &7E 126	&FC 252	&11 17	&E1 225	&01 1
" &22 34	&FC 252	Q &51 81	&09 9	A &41 65
2 &32 50	&FC 252	q &71 113	&09 9	a &61 97

70 CAPS LOCK	71 Z *	72 [▲] *	73 [▼] *	74 [◀] *
&FE 254	&1A 26	&FF 255	&FF 255	&FF 255
&FD 253	Z &5A 90	&0B 11	&0A 10	&08 8
&FD 253	z &7A 122	&0B 11	&0A 10	&08 8
75 [▶] *	**76 [FEUER 2]**	**77 [FEUER 1]**	**78 ex.nicht**	**79 DEL ***
&FE 254	&FE 254	&FE 254	&FE 254	&FE 254
&FD 253	X &FD 253	Z &FD 253	&FD 253	&FD 253
&FD 253	x &FD 253	Z &FD 253	&FD 253	&FD 253

ERWEITERUNGSZEICHEN

Der Key-Manager bietet die Möglichkeit, 32 Zeichenketten zu definieren, die den Zeichen 128 bis 159 zugeordnet werden. Standardmäßig ist der Zehnerblock damit belegt.

Taste	Erweit.zeichen	zugeordneter String			
ENTER	solo:	&8B	139	→	CHR$(13)
	SHIFT:	&8C	140	→	CHR$(13)
	CTRL:	&8C	140	→	RUN" + CHR$(13)
0	s/s/c:	&80	128	→	0
1	s/s/c:	&81	129	→	1
2	s/s/c:	&82	130	→	2
3	s/s/c:	&83	131	→	3
4	s/s/c:	&84	132	→	4
5	s/s/c:	&85	133	→	5
6	s/s/c:	&86	134	→	6
7	s/s/c:	&87	135	→	7
8	s/s/c:	&88	136	→	8
9	s/s/c:	&89	137	→	9
.	s/s/c:	&8A	138	→	.

STEUERZEICHEN DES KEY-MANAGERS UND DES ZEILENEDITORS

Einige Zeichen erfahren durch den Key-Manager eine Sonderbehandlung und lassen sich deshalb durch die Tastatur normalerweise nicht erzeugen. Noch viel mehr Zeichen werden aber vom Zeileneditor als Steuerzeichen benutzt und sind deshalb nicht direkt eingebbar.

Zeichen	Bedeutung	Standardtaste
Sonderbehandlung durch den Key-Manager		
&FF 255	kein Zeichen (ignorieren)	z.B. 1 plus CTRL
&FE 254	Flip Shift-Lock	CAPS LOCK plus CTRL
&FD 253	Flip Caps-Lock	CAPS LOCK
&80 128 bis &9F 159	Erweiterungszeichen	Zeichen 128 bis 140 auf dem Zehnerblock
Sonderbehandlung durch den Zeileneditor:		
&00 = 0	kein Zeichen (ignorieren)	@ plus CTRL
&0D = 13	ENTER: Zeile übernehmen	ENTER
&10 = 16	lösche Zeichen auf Crsrpos.	CLR
&7F = 127	DELETE:lösche Zeich vor Crsr	DEL
&E0 = 224	kopiere Zeichen vom Copy-Crsr	COPY
&E1 = 225	Flip Insert-Flag TAB plus	CTRL
&EF = 239	kein Zeichen (ignorieren)	Break-Event-Token
&F0 = 240 bis &FB = 251	Cursor und Copycursor bewegen Crsr → Text/Zeilen-Anfang Crsr → Text/Zeilen-Ende	Cursortasten solo, mit SHIFT und mit CTRL
&FC =252	* BREAK *	ESC

Anhang F

Die Bildausgabe

Die Controlcodes

Versucht man, Zeichen mit dem Code von 0 bis 31 auf dem Bildschirm auszudrucken, so werden diese normalerweise als Controlcodes interpretiert und nicht dargestellt, sondern befolgt. Die Standardfunktionen finden sich hier.

Die mit * gekennzeichneten Codes zwingen den Cursor vor ihrer Ausführung auf eine legale Position im Textfenster. Die mit # gekennzeichneten Codes sind nicht mit dem Zeileneditor von BASIC eingebbar (aber mit dem Copy-Cursor außer CHR$(0)).

Beim CPC 464 werden alle Controlcodes auch dann befolgt, wenn die Textausgabe im aktuellen Textfenster "disabled" ist (Vektor &BB57 oder CHR$(21) = NAK), beim CPC 664 und 6128 jedoch nur solche Codes, bei denen ein Flag entsprechend gesetzt ist. Die Codes, die beim CPC 664 und CPC 6128 nicht mehr ausgeführt werden, wenn der Textstrom disabled ist, sind mit ! gekennzeichnet.

[CTRL] +Taste	Code hex	dez	ASCII-Name	Parameter	Standardfunktion
! [@]	&00	00	NUL	–	wird ignoriert.
! [A]	&01	01	SOH	c	Das Zeichen mit dem Code c wird gedruckt. Damit sind die Sonderzeichen 0 bis 31 darstellbar.
! [B]	&02	02	STX	–	Cursor ausschalten (User-Ebene).
! [C]	&03	03	ETX	–	Cursor wieder einschalten.
! [D]	&04	04	EOT	m	Bildschirm auf Modus m umstellen; m wird mit &3 maskiert.
! [E]	&05	05	ENQ	c	Das Zeichen mit dem Code c wird auf der Position des Grafik-Cursors ausgegeben.

[CTRL] +Taste	Code hex	dez	ASCII-Name	Parameter	Standardfunktion
[F]	&06	06	ACK	–	Ausgabe von Zeichen in diesem Textfenster wieder zulassen (→ &15).
! [G]	&07	07	BEL	–	Warnpiepser ausgeben.
! * [H]	&08	08	BS	–	Cursor um eine Position zurück.
! * [I]	&09	09	TAB	–	Cursor um eine Position weiter vor.
! * [J]	&0A	10	LF	–	Cursor um eine Zeile nach unten.
! * [K]	&0B	11	VT	–	Cursor um eine Zeile nach oben.
! [L]	&0C	12	FF	–	Cursor in die linke obere Ecke setzen und den Bildschirm löschen.
! #* [M]	&0D	13	CR	–	Cursor in die erste Spalte der aktuellen Cursor-Zeile setzen.
! [N]	&0E	14	SO	p	Hintergrund-Tinte des aktuellen Textfensters auf p festlegen; p wird mit &F maskiert.
! [O]	&0F	15	SI	p	Vordergrund-Tinte des aktuellen Textfensters auf p festlegen. p wird mit &F maskiert.
! #* [P]	&10	16	DLE	–	Löschen des Zeichens auf der Cursorposition.
! * [Q]	&11	17	DC1	–	Cursor-Zeile bis zur Cursor-Spalte löschen, inkl. der Cursor-Position.
! * [R]	&12	18	DC2	–	Cursor-Zeile ab der Cursor-Spalte löschen, inkl. der Cursor-Position.
! * [S]	&13	19	DC3	–	Textfenster bis inkl. Cursor-Position löschen.
! * [T]	&14	20	DC4	–	Textfenster ab inkl. Cursor-Postion löschen.
! [U]	&15	21	NAK	–	Das Ausdrucken von Zeichen im aktuellen Textfenster verbieten (→ &6).
! [V]	&16	22	SYN	t	Den Transparentmodus einschalten (t=1) oder auschalten (t=0); t wird mit &1 maskiert.

[CTRL] +Taste	Code hex	dez	ASCII-Name	Parameter	Standardfunktion
! [W]	&17	23	ETB	g	Den Grafikmodus auf g festlegen; g wird mit &3 maskiert. g=0 – Tinte ist deckend, opaque, normal g=1 – X-Odern der neuen & alten Tinte g=2 – Und-Verknüpfung von n.& a. Tinte g=3 – Odern
! [X]	&18	24	CAN	–	Austauschen der Vorder- und Hintergrund-Tinten.
! [Y]	&19	25	EM	zmmmmmmmm	Die Zeichenmatrix für das Zeichen z neu definieren. Die 8 Parameter m codieren mit jeweils 8 Bits die neue 8*8-Matrix für das Zeichen.
! [Z]	&1A	26	SUB	lrou	Die Grenzen des aktuellen Textfensters neu festlegen und den Cursor in die linke obere Ecke setzen. l und r geben den linken und rechten Rand an, o und u den oberen und unteren l und r bzw. o und u werden automatisch nach Größe sortiert und evtl. vertauscht.
[[]	&1B	27	ESC	–	wird ignoriert.
! [\]	&1C	28	FS	tff	Die Blinkfarben der Tinte t werden mit den beiden Farben f neu festgelegt. t wird mit &F und f jeweils mit &1F maskiert.
! []]	&1D	29	GS	ff	Die Blinkfarben für die Bildschirmumrandung werden mit den beiden Farben f festgelegt. Diese werden jeweils mit &1F maskiert.
! [^]	&1E	30	RS	–	Den Cursor in die linke obere Ecke setzen.
! [0]	&1F	31	US	sz	Den Cursor in Spalte s und Zeile z setzen.

Der CRTC HD 6845

OUT (&BCxx) – Register adressieren (anwählen)
OUT (&BDxx) – Register beschreiben
INP (&BExx) – reserviert für Status lesen
INP (&BFxx) – Register einlesen

r = lesbares / w = beschreibbares Register Werte rechts: Standardeinstellung			NTSC dez/hex		PAL/SECAM dez/hex	
R00:	(–/w)	theoretische Zeichenzahl für eine Zeile inkl. Border & Strahlrücklauf	63	&3F	63	&3F
R01:	(–/w)	dargestellte Zeichen pro Zeile	40	&28	40	&28
R02:	(–/w)	Zeitpunkt für horizontale Synchr.	46	&2E	46	&2E
R03:	(–/w)	Breite des horizontalen Synchr.Pulses	142	&8E	142	&8E
R04:	(–/w)	theoretische Zeichenzahl für eine Spalte inkl. Border & Strahlhochlauf	31	&1F	38	&26
R05:	(–/w)	Feinabgleich zu R04	06	&06	00	&00
R06:	(–/w)	dargestellte Zeichen pro Spalte	25	&19	25	&19
R07:	(–/w)	Zeitpunkt für vertikale Synchr.	27	&1B	30	&1E
R08:	(–/w)	Schalter für Zeilensprung-Verfahren	00	&00	00	&00
R09:	(–/w)	Rasterzeilen/Buchstabe – 1	07	&07	07	&07
R10:	(–/w)	Einstellung für Hardware-Cursor	00	&00	00	&00
R11:	(–/w)	Einstellung für Hardware-Cursor	00	&00	00	&00
R12:	(r/w)	Text-Startadresse (msb)	48	&30	48	&30
R13:	(r/w)	Text-Startadresse (lsb)	00	&00	00	&00
R14:	(r/w)	Adresse des Hardware-Cursors (msb)	192	&C0	192	&C0
R15:	(r/w)	Adresse des Hardware-Cursors (lsb)	00	&00	00	&00
R16:	(r/)	Lightpen-Position (msb)	--		--	
R17:	(r/)	Lightpen-Position (lsb)	--		--	

Adressierung des Video-RAMs:

CPU-Adresse		CRTC-Adresse
A0		– von der ULA
A1	bis A10	– MA0 bis MA9
A11	bis A13	– RA0 bis RA2
A14	und A15	– MA12 und MA13

TINTEN UND FARBEN

Beim Schneider CPC kann man jeder Tinte zwei Farben zuordnen. Diese werden dann abwechselnd mit jeder Blinkperiode dargestellt.

Diese Zuordnung ist dabei recht aufwendig gestaltet. Zunächst muß man sich für zwei Farben entscheiden, die man einer Tinte zuordnen will (meist zwei

gleiche, damit's nicht blinkt). Diese Farben haben Farbnummern, die man nun dem Screen Pack zusammen mit der Tintennummer übermittelt.

Das Screen Pack übersetzt die Farbnummern in Paletten-Farbnummern, mit denen es dann im Blinkrhythmus das Paletten-RAM im Gate Array programmiert. Diese Umsetzung dient dazu, die Farbnummern nach steigender Helligkeit auf einem Grünmonitor zu sortieren.

Das Gate Array wandelt dann bei der Bildwiedergabe die Tintennummern, die in den Bytes des Bildwiederholspeichers codiert sind, in die gewünschten Farbsignale für den Monitorausgang um.

Standard-Zuordnung der Tinten zu Farbnummern

Tinte:	Border	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Farbe 1:	1	1	24	20	6	26	0	2	8	10	12	14	16	18	22	1	16
Farbe 2:	1	1	24	20	6	26	0	2	8	10	12	14	16	18	22	24	11

<—— Mode 2 ————> <——>
<—— Mode 1 ————> blinkend
<—— Mode 0 ————————————————————>

Zusammenhang zwischen Farbe, Farbnummer und Paletten-Farbnummer

Farbe	Paletten-Farbnr.	Farb-nummer	Farb-nummer	Paletten-Farbnr.	Farbe
Schwarz	20	0	26	11	Hellweiß
Blau	4	1	25	3	Pastellgelb
Hellblau	21	2	24	10	Hellgelb
Rot	28	3	23	27	Pastellblaugrün
Magenta	24	4	22	25	Pastellgrün
Hellviolett	29	5	21	26	Limonengrün
Hellrot	12	6	20	19	Hellblaugrün
Purpur	5	7	19	2	Seegrün
Hellmagenta	13	8	18	18	Hellgrün
Grün	22	9	17	15	Pastellmagenta
Blaugrün	6	10	16	7	Rosa

Farbe	Paletten-Farbnr.	Farb-nummer	Farb-nummer	Paletten-Farbnr.	Farbe
Himmelblau	23	11	15	14	Orange
Gelb	30	12	14	31	Pastellblau
Weiß	0	13	13	0	Weiß

Die ULA

Das Gate Array enthält Register, die die Bildschirmdarstellung und die ROM-Konfiguration beeinflussen. Durch Programmieren dieser Register kann man festlegen, welcher Bildschirmmodus dargestellt wird, welche Farben die einzelnen Tinten haben und ob oben oder unten ein ROM eingeblendet werden soll.

Port-Adresse: OUT &7FFF

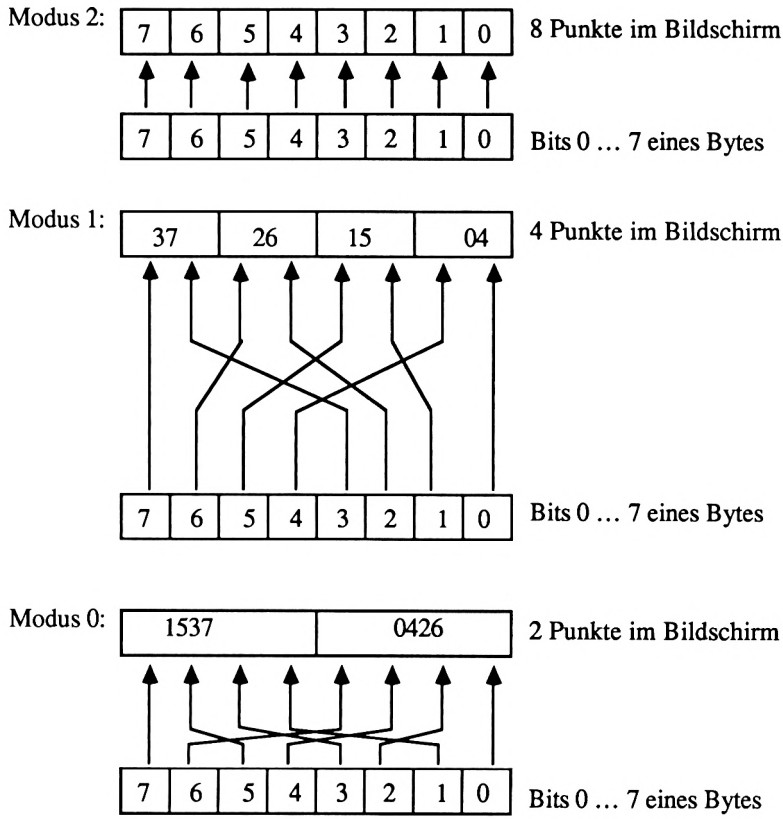
Datenbyte	Bedeutung der einzelnen Bits im Datenbyte
&X0000iiii	Wählt das Farbregister für Tinte iiii an
&X0001????	Wählt das BORDER-Farbregister an
&X010nnnnn	Programmiert das gerade angewählte Farbregister mit der Farbe nnnnn (Paletten-Farbnummer)
&X100roumm	r=1 – Löscht den 52-Bildschirmzeilen-Zähler (→ Interrupt-Verzögerung) o=1 – Blendet oberes ROM aus u=1 – Blendet unteres ROM aus mm – Bestimmt Bildschirmmodus

Anmerkung: Bit 5 ist immer 0 (reserviert). Das Z80-Registerpaar B'C' enthält ständig die Port-Adresse und das passende Datenbyte, um mit OUT (C),C die momentane Bank-Konfiguration und den Bildschirmmodus einzustellen.

Die Codierung der Tintennummern in den Bildschirm-Bytes:

In den folgenden Grafiken ist der Zusammenhang zwischen einem Byte und den Tintennummern der darin enthaltenen Pixels dargestellt. Der obere Balken symbolisiert die dargestellten Pixels, der untere Balken jeweils ein Byte

mit 8 Bits. Die Ziffergruppe "in den Pixeln" gibt dabei jeweils an, wie welche Bits des Bytes gruppiert werden müssen, um die Tintennummer dieses Pixels zu bestimmen.



PIXELMASKEN

Da diese Zuordnungen auch für die Grafik- und Textroutinen nur schwer zu behandeln sind, benutzt das Betriebssystem Tabellen dafür.

Eine davon enthält Bytes, die immer ein Pixel aus einem Byte herausfiltern:

Modus 2: Pixel 7 → Byte: &X10000000 = &80 = 128 (links)
Pixel 6 → Byte: &X01000000 = &40 = 64

Modus 2:	Pixel	5 →	Byte:	&X00100000	=	&20	= 32	
	Pixel	4 →	Byte:	&X00010000	=	&10	= 16	
	Pixel	3 →	Byte:	&X00001000	=	&08	= 8	
	Pixel	2 →	Byte:	&X00000100	=	&04	= 4	
	Pixel	1 →	Byte:	&X00000010	=	&02	= 2	
	Pixel	0 →	Byte:	&X00000001	=	&01	= 1	(rechts)
Modus 1:	Pixel	3 →	Byte:	&X10001000	=	&88	= 136	(links)
	Pixel	2 →	Byte:	&X01000100	=	&44	= 68	
	Pixel	1 →	Byte:	&X00100010	=	&22	= 34	
	Pixel	0 →	Byte:	&X00010001	=	&11	= 17	(rechts)
Modus 0:	Pixel	1 →	Byte:	&X10101010	=	&AA	= 170	(links)
	Pixel	0 →	Byte:	&X01010101	=	&55	= 85	(rechts)

FARBMASKEN (ENCODED INKS)

Bei jeder Zuordnung von Tinten zu Vorder- oder Hintergrund für die Text- und Grafikausgabe wird ein Byte berechnet, das vollständig mit dieser Tinte "eingefärbt" ist:

Modus 2:	Tinte	00 →	&X00000000	=	&00	= 0
	Tinte	01 →	&X11111111	=	&FF	= 255
Modus 1:	Tinte	00 →	&X00000000	=	&00	= 0
	Tinte	01 →	&X11110000	=	&F0	= 240
	Tinte	02 →	&X00001111	=	&0F	= 15
	Tinte	03 →	&X11111111	=	&FF	= 255
Modus 0:	Tinte	00 →	&X00000000	=	&00	= 0
	Tinte	01 →	&X11000000	=	&C0	= 192
	Tinte	02 →	&X00001100	=	&0C	= 12
	Tinte	03 →	&X11001100	=	&CC	= 204
	Tinte	04 →	&X00110000	=	&30	= 48
	Tinte	05 →	&X11110000	=	&F0	= 240
	Tinte	06 →	&X00111100	=	&3C	= 60
	Tinte	07 →	&X11111100	=	&FC	= 252
	Tinte	08 →	&X00000011	=	&03	= 3
	Tinte	09 →	&X11000011	=	&C3	= 195
	Tinte	10 →	&X00001111	=	&0F	= 15
	Tinte	11 →	&X11001111	=	&CF	= 207

Tinte 12 → &X00110011 = &33 = 51
Tinte 13 → &X11110011 = &F3 = 243
Tinte 14 → &X00111111 = &3F = 63
Tinte 15 → &X11111111 = &FF = 255

Der Zeichensatz des Schneider CPC

Die Zeichenausgabe des Schneider CPC auf dem Monitor ist voll Grafik-orientiert, das heißt, im Bildwiederholspeicher gibt es nur noch Informationen darüber, mit welcher Tinte jeder Punkt dargestellt werden soll. Die Textausgabe-Routinen im Betriebssystem müssen praktisch die Buchstaben in den Bildschirmspeicher "malen". Das Betriebssystem-ROM enthält deshalb eine Tabelle für insgesamt 256 verschiedene Zeichen. Für jedes Zeichen ist eine 8*8 Punkte große Matrix angelegt, die in Form von 8 Bytes zu je 8 Bits gespeichert ist. Teile oder auch der komplette Zeichensatz können aus dem unteren ROM ins RAM kopiert werden, um hier veränderbar zu sein. Das wurde im folgenden Programm ausgenutzt, um die danach folgenden Tabelle auszudrucken:

```
130 CLOSEIN:CLOSEOUT
140 SYMBOL AFTER 0:start=HIMEM+1
150 str=9:IF str=9 THEN OPENOUT"zeichen.dat
151 '
152 PRINT#str,"Der Zeichensatz der Schneider CPCs:"
153 PRINT#str,"-----------------------------------"
160 '
170 nbn=8:ZONE 10
180 '
190 FOR i=0 TO (256-nbn)*8 STEP 8*nbn
200   PRINT#str
205   PRINT#str,"&";HEX$(i\8,2);" =";i\8;":"
209   PRINT#str
210   FOR j=0 TO 7
220     FOR adr=start+i+j TO start+i+j+(nbn-1)*8 STEP 8
230       b$=BIN$(PEEK(adr),8):GOSUB 330
240     NEXT
250     IF POS(#str)>1 THEN PRINT#str
260   NEXT
270 NEXT
280 CLOSEOUT
290 END
300 '
310 ' print b$
320 '
330 FOR p=1 TO 8
340   IF MID$(b$,p,1)="0" THEN PRINT#str,"+"; ELSE PRINT#str,"#";
350 NEXT
360 PRINT#str,,
370 RETURN
```

Der Zeichensatz des Schneider CPC:

&00 = 0 :

```
MMMMMMMM  MMMMMMMM  ---MM---  ------MM  ----MM--  MMMMMMMM  --------  --MMMM--
MM----MM  MM------  ---MM---  ------MM  ---MM---  MM----MM  -------M  -MM--MM-
MM----MM  MM------  ---MM---  ------MM  --MM----  MMM--MMM  ------MM  MM----MM
MM----MM  MM------  ---MM---  ------MM  -MMMMMM-  MM-MM-MM  -----MM-  MM----MM
MM----MM  MM------  ---MM---  ------MM  ----MM--  MM-MM-MM  MM--MM--  MMMMMMMM
MM----MM  MM------  ---MM---  ------MM  ---MM---  MMM--MMM  -MMMM---  --M--M--
MM----MM  MM------  ---MM---  ------MM  --MM----  MM----MM  --MM----  MMM--MMM
MMMMMMMM  MM------  MMMMMMMM  MMMMMMMM  --------  MMMMMMMM  --------  --------
```

&08 = 8 :

```
--------  --------  ---MM---  ---MM---  ---MM---  --------  --MMMM--  --MMMM--
--------  --------  ---MM---  --MMMM--  -M-MM-M-  ------MM  -MM--MM-  -MM--MM-
--MM----  ----MM--  ---MM---  -MMMMMM-  --MMMM--  --MM--MM  MMMMMMMM  MM----MM
-MM-----  -----MM-  ---MM---  MM-MM-MM  M--MM--M  -MM---MM  MM-MM-MM  MM-MM-MM
MMMMMMMM  MMMMMMMM  MM-MM-MM  ---MM---  MM-MM-MM  MMMMMMM-  MM-MM-MM  MM-MM-MM
-MM-----  -----MM-  -MMMMMM-  ---MM---  -MMMMMM-  -MM-----  MMMMMMMM  MM----MM
--MM----  ----MM--  --MMMM--  ---MM---  --MMMM--  --MM----  -MM--MM-  -MM--MM-
--------  --------  ---MM---  ---MM---  ---MM---  --------  --MMMM--  --MMMM--
```

&10 = 16 :

```
MMMMMMMM  --MMMM--  --MMMM--  --MMMM--  --MMMM--  --------  -MMMMMM-  ------MM
MM----MM  -MMMMMM-  -MM--MM-  -MM--MM-  -MMMMMM-  -------M  -MM--MM-  ------MM
MM----MM  MM-MM-MM  MM----MM  MM----MM  MM-MM-MM  --MM--MM  -MM--MM-  ------MM
MMMMMMMM  MM-MM-MM  MM-MMMMM  MMMMM-MM  MM-MM-MM  ---MMMM-  -MM--MM-  MMMMMMMM
MM----MM  MM-MMMMM  MM-MM-MM  MM-MM-MM  MMMMM-MM  MM--MMM-  -MM--MM-  ------MM
MM----MM  MM----MM  MM-MM-MM  MM-MM-MM  MM----MM  -MMMM-MM  -MM--MM-  ------MM
MM----MM  -MM--MM-  -MMMMMM-  -MMMMMM-  -MM--MM-  --MM---M  -MM--MM-  ------MM
MMMMMMMM  --MMMM--  --MMMM--  --MMMM--  --MMMM--  --------  MMM--MMM  --------
```

&18 = 24 :

```
MMMMMMMM  ---MM---  --MMMM--  --MMMM--  MMMMMMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM
-MM--MM-  ---MM---  -MM--MM-  -MM--MM-  MM-MM-MM  MM----MM  MM----MM  MM-MM-MM
--MMMM--  --MMMM--  -MM--MM-  MM----MM  MM-MM-MM  MM----MM  MM----MM  MM-MM-MM
---MM---  --MMMM--  --MM----  MMMMMMMM  MM-MM-MM  MMMMM-MM  MM-MMMMM  MM-MM-MM
---MM---  --MMMM--  ---MM---  MM----MM  MMMMM-MM  MM-MM-MM  MM-MM-MM  MM-MMMMM
--MMMM--  --MMMM--  --------  MM----MM  MM----MM  MM-MM-MM  MM-MM-MM  MM----MM
-MM--MM-  ---MM---  ---MM---  -MM--MM-  MM----MM  MM-MM-MM  MM-MM-MM  MM----MM
MMMMMMMM  ---MM---  --------  --MMMM--  MMMMMMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM
```

&20 = 32 :

```
--------  ---MM---  -MM-MM--  -MM-MM--  ---MM---  --------  --MMM---  ---MM---
--------  ---MM---  -MM-MM--  -MM-MM--  --MMMM-  MM---MM-  -MM-MM--  ---MM---
--------  ---MM---  -MM-MM--  MMMMMMM-  -M-MM---  MM--MM--  --MMM---  --MM----
--------  ---MM---  --------  -MM-MM--  --MMM--  ---MM---  -MMM-MM-  --------
--------  ---MM---  --------  MMMMMMM-  ---MM-M-  --MM----  MM-MMM--  --------
--------  --------  --------  -MM-MM--  -MMMM--  -MM--MM-  MM--MM--  --------
--------  ---MM---  --------  -MM-MM--  ---MM---  MM---MM-  -MMM-MM-  --------
--------  --------  --------  --------  --------  --------  --------  --------
```

&28 = 40 :

```
----MM--  --MM----  --------  --------  --------  --------  --------  -----MM-
---MM---  ---MM---  -MM--MM-  ---MM---  --------  --------  --------  ----MM--
--MM----  ----MM--  --MMMM--  ---MM---  --------  --------  --------  ---MM---
--MM----  ----MM--  MMMMMMMM  -MMMMM-  --------  -MMMMM-  --------  --MM----
--MM----  ----MM--  --MMMM--  ---MM---  --------  --------  --------  -MM-----
---MM---  ---MM---  -MM--MM-  ---MM---  ---MM---  --------  ---MM---  MM------
----MM--  --MM----  --------  --------  ---MM---  --------  ---MM---  M-------
--------  --------  --------  --------  --MM----  --------  --------  --------
```

&30 = 48 :

```
-MMMM--  ---MM---  --MMM--  --MMM--  ---MM--  -MMMMM-  --MMM--  -MMMMM-
MM--MM-  --MMM---  -MM--MM-  -MM--MM-  --MMM--  -MM---M-  -MM--MM-  -MM--MM-
MM--MM-  ---MM---  -----MM-  -----MM-  -MM-MM--  -MM-----  -MM-----  -----MM-
MM-M-MM-  ---MM---  --MMM--  ---MM--  MM--MM--  -MMMM--  -MMMM--  ----MM--
MMM--MM-  ---MM---  -MM-----  -----MM-  MMMMMM-  -----MM-  -MM--MM-  ---MM---
MM--MM-  ---MM---  -MM--MM-  -MM--MM-  ----MM--  -MM--MM-  -MM--MM-  ---MM---
-MMMM--  -MMMMM-  -MMMMM-  --MMM--  ---MMM-  --MMM--  --MMM--  ---MM---
--------  --------  --------  --------  --------  --------  --------  --------
```

&38 = 56 :

```
--MMM--  --MMM--  --------  --------  ----MM--  --------  -MM-----  --MMM--
-MM--MM-  -MM--MM-  --------  --------  ---MM---  --------  --MM----  -MM--MM-
-MM--MM-  -MM--MM-  ---MM---  ---MM---  --MM----  -MMMMM-  ---MM---  -MM--MM-
--MMM--  --MMMM-  ---MM---  ---MM---  -MM-----  --------  ----MM--  ----MM--
-MM--MM-  -----MM-  --------  --------  --MM----  --------  ---MM---  ---MM---
-MM--MM-  -MM--MM-  ---MM---  ---MM---  ---MM---  -MMMMM-  --MM----  --------
--MMM--  --MMM--  ---MM---  ---MM---  ----MM--  --------  -MM-----  ---MM---
--------  --------  --------  --MM----  --------  --------  --------  --------
```

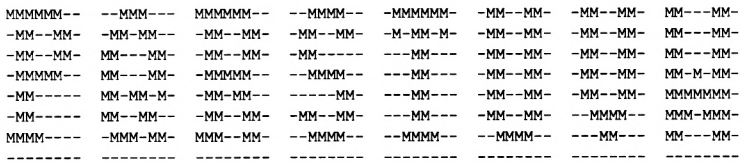
&40 = 64 :

```
-MMMM--  ---MM---  MMMMM--  --MMM--  MMMM---  MMMMMM-  MMMMMM-  --MMM--
MM--MM-  --MMM--  -MM--MM-  -MM--MM-  -MM-MM--  -MM---M-  -MM---M-  -MM--MM-
MM-MMM-  -MM--MM-  -MM--MM-  MM------  -MM--MM-  -MM-M---  -MM-M---  MM------
MM-MMM-  -MM--MM-  -MMMM--  MM------  -MM--MM-  -MMM----  -MMMM---  MM------
MM-MMM-  -MMMMM-  -MM--MM-  MM------  -MM--MM-  -MM-M---  -MM-M---  MM--MM-
MM------  -MM--MM-  -MM--MM-  -MM--MM-  -MM-MM--  -MM---M-  -MM-----  -MM--MM-
-MMMM--  -MM--MM-  MMMMM--  --MMM--  MMMM---  MMMMMM-  MMMM----  --MMMM-
--------  --------  --------  --------  --------  --------  --------  --------
```

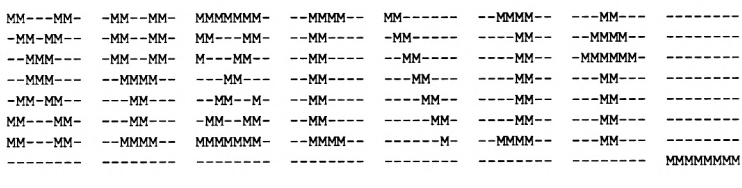
&48 = 72 :



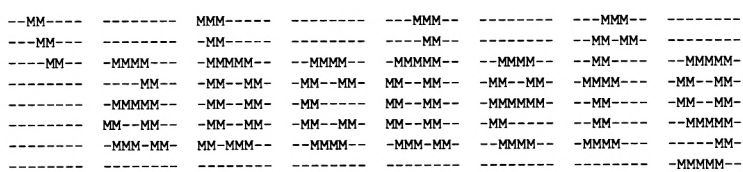
&50 = 80 :



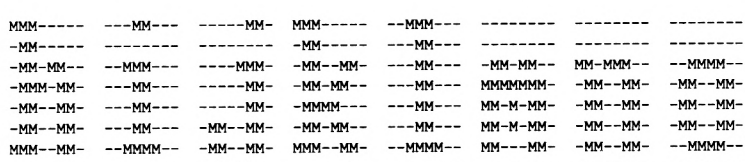
&58 = 88 :



&60 = 96 :



&68 = 104 :



&70 = 112 :

```
--------  --------  --------  --------  --MM----  --------  --------  --------
--------  --------  --------  --------  --MM----  --------  --------  --------
MM-MMM--  -MMM-MM-  MM-MMM--  --MMMM--  -MMMMM--  -MM--MM-  -MM--MM-  MM---MM-
-MM--MM-  MM--MM--  -MMM-MM-  -MM-----  --MM----  -MM--MM-  -MM--MM-  MM-M-MM-
-MM--MM-  MM--MM--  -MM-----  --MMMM--  --MM----  -MM--MM-  -MM--MM-  MM-M-MM-
-MMMMM--  -MMMMM--  -MM-----  -----MM-  --MM-MM-  -MM--MM-  --MMMM--  MMMMMMM-
-MM-----  ----MM--  MMMM----  -MMMMM--  ---MMM--  --MMMMM-  ---MM---  -MM-MM--
MMMM----  ---MMMM-  --------  --------  --------  --------  --------  --------
```

&78 = 120 :

```
--------  --------  --------  ----MMM-  ---MM---  -MMM----  -MMM-MM-  MM--MM--
--------  --------  --------  ---MM---  ---MM---  ---MM---  MM-MMM--  --MM--MM
MM---MM-  -MM--MM-  -MMMMMM-  ---MM---  ---MM---  ---MM---  --------  MM--MM--
-MM-MM--  -MM--MM-  -M--MM--  -MMM----  ---MM---  ----MMM-  --------  --MM--MM
--MMM---  -MM--MM-  ---MM---  ---MM---  ---MM---  ---MM---  --------  MM--MM--
-MM-MM--  --MMMMM-  --MM--M-  ---MM---  ---MM---  ---MM---  --------  --MM--MM
MM---MM-  -----MM-  -MMMMMM-  ----MMM-  ---MM---  -MMM----  --------  MM--MM--
--------  -MMMMM--  --------  --------  --------  --------  --------  --MM--MM
```

&80 = 128 :

```
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  --------  --------  --------  MMMM----  MMMM----  MMMM----  MMMM----
--------  --------  --------  --------  MMMM----  MMMM----  MMMM----  MMMM----
--------  --------  --------  --------  MMMM----  MMMM----  MMMM----  MMMM----
--------  --------  --------  --------  MMMM----  MMMM----  MMMM----  MMMM----
```

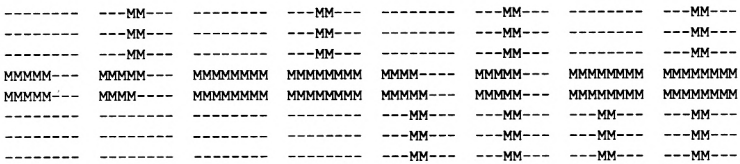
&88 = 136 :

```
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
--------  MMMM----  ----MMMM  MMMMMMMM  --------  MMMM----  ----MMMM  MMMMMMMM
----MMMM  ----MMMM  ----MMMM  ----MMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM
----MMMM  ----MMMM  ----MMMM  ----MMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM
----MMMM  ----MMMM  ----MMMM  ----MMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM
----MMMM  ----MMMM  ----MMMM  ----MMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM  MMMMMMMM
```

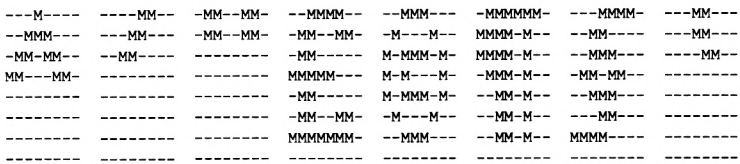
&90 = 144 :

```
--------  ---MM---  --------  ---MM---  --------  ---MM---  --------  ---MM---
--------  ---MM---  --------  ---MM---  --------  ---MM---  --------  ---MM---
--------  ---MM---  --------  ---MM---  --------  ---MM---  --------  ---MM---
---MM---  ---MM---  ---MMMMM  ---MMMMM  ---MM---  ---MM---  ----MMMM  ---MMMMM
---MM---  ---MM---  ---MMMMM  ----MMMM  ---MM---  ---MM---  ---MMMMM  ---MMMMM
--------  --------  --------  --------  ---MM---  ---MM---  ---MM---  ---MM---
--------  --------  --------  --------  ---MM---  ---MM---  ---MM---  ---MM---
--------  --------  --------  --------  ---MM---  ---MM---  ---MM---  ---MM---
```

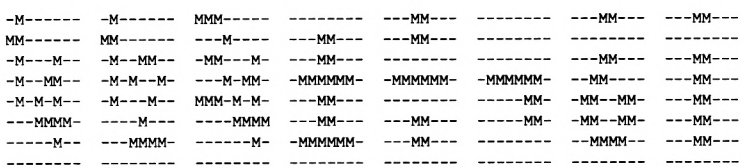
&98 = 152 :



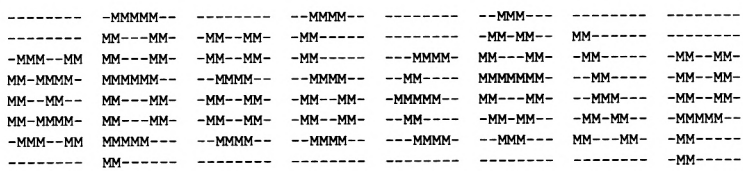
&A0 = 160 :



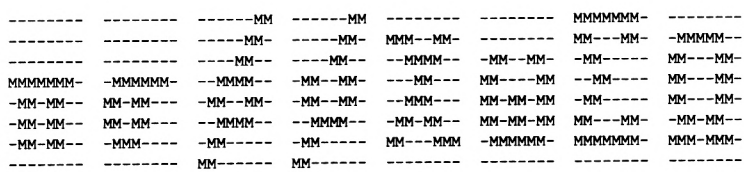
&A8 = 168 :



&B0 = 176 :



&B8 = 184 :



&C0 = 192 :

```
---MM---  ---MM---  --------  --------  ---MM---  ---MM---  --------  ---MM---
--MM----  ----MM--  --------  --------  --MMMM--  ----MM--  --------  --MM----
-MM-----  -----MM-  --------  --------  -MM--MM-  -----MM-  --------  -MM-----
MM------  ------MM  -------M  M-------  MM----MM  ------MM  M------M  MM------
M-------  -------M  ------MM  MM------  M------M  ------MM  MM----MM  MM------
--------  --------  -----MM-  -MM-----  --------  -----MM-  -MM--MM-  -MM-----
--------  --------  ----MM--  --MM----  --------  ----MM--  --MMMM--  --MM----
--------  --------  ---MM---  ---MM---  --------  ---MM---  ---MM---  ---MM---
```

&C8 = 200 :

```
---MM---  ---MM---  ---MM---  MM----MM  ------MM  MM------  MM--MM--  M-M-M-M-
--MM----  ----MM--  --MMMM--  MMM--MMM  -----MMM  MMM-----  MM--MM--  -M-M-M-M
-MM-----  -----MM-  -MM--MM-  -MMMMMM-  ----MMM-  -MMM----  --MM--MM  M-M-M-M-
MM-----M  M-----MM  MM----MM  --MMMM--  ---MMM--  --MMM---  --MM--MM  -M-M-M-M
M-----MM  MM-----M  MM----MM  --MMMM--  --MMM---  ---MMM--  MM--MM--  M-M-M-M-
-----MM-  -MM-----  -MM--MM-  -MMMMMM-  -MMM----  ----MMM-  MM--MM--  -M-M-M-M
----MM--  --MM----  --MMMM--  MMM--MMM  MMM-----  -----MMM  --MM--MM  M-M-M-M-
---MM---  ---MM---  ---MM---  MM----MM  MM------  ------MM  --MM--MM  -M-M-M-M
```

&D0 = 208 :

```
MMMMMMMM  ------MM  --------  MM------  MMMMMMMM  MMMMMMMM  -------M  M-------
MMMMMMMM  ------MM  --------  MM------  MMMMMMM-  -MMMMMMM  ------MM  MM------
--------  ------MM  --------  MM------  MMMMMM--  --MMMMMM  -----MMM  MMM-----
--------  ------MM  --------  MM------  MMMMM---  ---MMMMM  ----MMMM  MMMM----
--------  ------MM  --------  MM------  MMMM----  ----MMMM  ---MMMMM  MMMM----
--------  ------MM  --------  MM------  MMM-----  -----MMM  --MMMMMM  MMMMM---
--------  ------MM  MMMMMMMM  MM------  MM------  ------MM  -MMMMMMM  MMMMMM--
--------  ------MM  MMMMMMMM  MM------  M-------  -------M  MMMMMMMM  MMMMMMMM
```

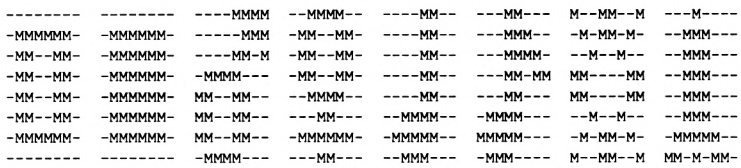
&D8 = 216 :

```
M-M-M-M-  ----M-M-  --------  M-M-----  M-M-M-M-  M-M-M-M-  -------M  --------
-M-M-M-M  -----M-M  --------  -M-M----  -M-M-M--  -M-M-M-M  ------M-  M-------
M-M-M-M-  ----M-M-  --------  M-M-----  M-M-M---  --M-M-M-  -----M-M  -M------
-M-M-M-M  -----M-M  --------  -M-M----  -M-M----  ---M-M-M  ----M-M-  M-M-----
--------  ----M-M-  M-M-M-M-  M-M-----  M-M-----  ----M-M-  ---M-M-M  -M-M----
--------  -----M-M  -M-M-M-M  -M-M----  -M------  -----M-M  --M-M-M-  M-M-M---
--------  ----M-M-  M-M-M-M-  M-M-----  M-------  ------M-  -M-M-M-M  -M-M-M--
--------  -----M-M  -M-M-M-M  -M-M----  --------  -------M  M-M-M-M-  M-M-M-M-
```

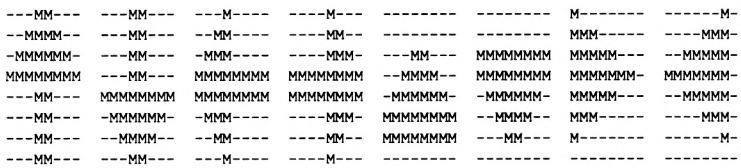
&E0 = 224 :

```
-MMMMMM-  -MMMMMM-  --MMM---  ---M----  -MM-MM--  ---M----  --------  --------
MMMMMMMM  MMMMMMMM  --MMM---  --MMM---  MMMMMMM-  --MMM---  --MMMM--  --MMMM--
M--MM--M  M--MM--M  MMMMMMM-  -MMMMM--  MMMMMMM-  -MMMMM--  -MM--MM-  -MMMMMM-
MMMMMMMM  MMMMMMMM  MMMMMMM-  MMMMMM--  MMMMMMM-  MMMMMM--  MM----MM  MMMMMMMM
M-MMMM-M  MM----MM  MMMMMMM-  -MMMMM--  -MMMMM--  MMMMMMM-  MM----MM  MMMMMMMM
MM----MM  M-MMMM-M  ---M----  --MMM---  --MMM---  ---M----  -MM--MM-  -MMMMMM-
MMMMMMMM  MMMMMMMM  --MMM---  ---M----  ---M----  --MMM---  --MMMM--  --MMMM--
-MMMMMM-  -MMMMMM-  --------  --------  --------  --------  --------  --------
```

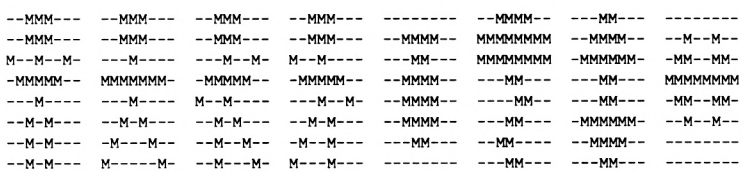
&E8 = 232 :



&F0 = 240 :



&F8 = 248 :



Anhang G

Die Tonausgabe

Ansteuerung des PSG

Der Sound-Generator hat keine eigene I/O-Adresse zugestanden bekommen, sondern wird vollständig über die PIO angesprochen. Hier ist er wie folgt angeschlossen:

Port-Adresse	PIO		PSG
&F4FF	Port A (I/O)	◄——►	Datenbus
&F6FF	Port C (-/O)	Bit 7 —►	BDIR
		Bit 6 —►	BC1

BC1	BDIR	Funktion:
0	0	Datenwort wird ignoriert
0	1	in adressiertes Register schreiben
1	0	aus adressiertem Register lesen
1	1	Register adressieren

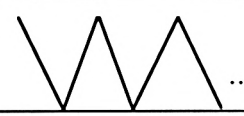
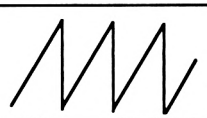
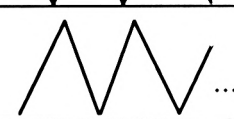
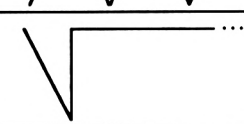
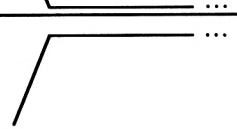
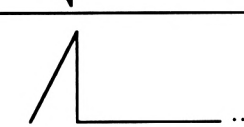
Das Kontrollregister (Reg. 7)

Bit	0	1	Bedeutung
0	ja	nein	Tonausgabe auf Kanal A
1	ja	nein	Tonausgabe auf Kanal B
2	ja	nein	Tonausgabe auf Kanal C
3	ja	nein	Rauschen auf Kanal A zumischen
4	ja	nein	Rauschen auf Kanal B zumischen
5	ja	nein	Rauschen auf Kanal C zumischen
6	in	out	Richtung des I/O-Ports

Die Register des AY-3-8912

Register	Belegung	Funktion
0	xxxxxxxx	LSB der Tonperiodenlänge
1	 xxxx	MSB für Kanal A
2	xxxxxxxx	LSB der Tonperiodenlänge
3	 xxxx	MSB für Kanal B
4	xxxxxxxx	LSB der Tonperiodenlänge
5	 xxxx	MSB für Kanal C
6	... xxxxx	Rauschperiodenlänge
7	. xxxxxxx	Kontrollregister
8	... hxxxx	Lautstärke Kanal A
9	... hxxxx	Lautstärke Kanal B
10	... hxxxx	Lautstärke Kanal C
11	xxxxxxxx	LSB der Periodenlänge des
12	xxxxxxxx	MSB Hüllkurvengenerators
13	 xxxx	Hüllkurvenform
14	xxxxxxxx	I/O-Port

Die möglichen Hüllkurvenformen (Register 13)

Nummer	Form	Nummer	Form
1000		1010	
1100		1110	
1001 00xx		1011	
1101		1111 01xx	

Periodenlängen der Noten aus 9 Oktaven

Die Frequenzen der untersten Oktave –4 bewegen sich bereits im Subsonic-Bereich. Andererseits ist Oktave +4 überhaupt nicht mehr brauchbar, da hier benachbarte Töne durch die Rundung der Periodenlänge teilweise schon auf dieselbe Note fallen. Auch Oktave 3 ist keinem mehr zuzumuten: Die einzelnen Töne sind bereits mit Fehlern von bis zu 25 % (bezogen auf den Halbtonschritt) behaftet.

Deshalb wird der PSG normalerweise auch mit der doppelten Frequenz, also 2 MHz, angesteuert. Dadurch würde die Subsonic-Oktave wegfallen, und die oberen Oktaven wären doppelt so genau. Oktave 3 wäre noch sehr gut und Oktave 4 so gut wie jetzt Oktave 3 angenähert.

Wer einen Blick ins CPC-464-Handbuch auf die dort wiedergegebenen Tabellen wirft, stellt fest, daß dort zu jeder Note die doppelte PSG-Periodenlänge angegeben ist. Das ist falsch, wäre aber für einen Eingangstakt von 2 MHz genau richtig. Möglicherweise hatte man also auch bei AMSTRAD zuerst 2 MHz vorgesehen und ist dann auf 1 MHz umgestiegen.

Die Tabelle wurde mit dem folgenden Programm erstellt:

```
10 OPENOUT"f":s=9                ' Mit s=0 Testlauf auf dem Bildschirm
20 freqa=440                     ' Frequenz des internationalen A
22 hts=2^(1/12)                  ' Halbtonschritt:
                                 ' Frequ.Verhältnis der Prime
25 DEF FNplen(i)=1000000/16/i
30 DIM n$(11):FOR i=0 TO 11:READ n$(i):NEXT
40 DATA C,CIS,D,DIS,E,F,FIS,G,GIS,A,AIS,H
50 '
60 FOR okt=-4 TO 4
70 PRINT#s:PRINT#s,"***** Oktave ";USING"+#&";okt;" *****":PRINT#s
75 PRINT#s,"Note   Frequenz   Periodenlänge   relativer Fehler in %"
77 PRINT#s,"        (Hertz)   genau    round   1/100
                                                1/Halbton/100":PRINT#s
80 FOR hton=0 TO 11
90 '
100 freq=freqa*2^(okt+(hton-9)/12)    ' Frequenz der Note
110 plen=FNplen(freq)                 ' Periodenlänge der Note (genau)
120 plen%=plen                        ' Periodenlänge der Note (gerundet)
130 note$=RIGHT$("    "+n$(hton),4)   ' Notenbezeichnung
140 fehler=(plen%-plen)/plen*100      ' Relativer Fehler in Prozent
150 '
160 PRINT#s,note$;" | ";USING"#####.##";freq;
170 PRINT#s," | ";USING"####.###&####";plen;"  ";plen%;
180 PRINT#s," | ";USING"+#.####&+##.####";fehler;"  ";fehler/(hts-1);
190 '
200 PRINT#s:NEXT
210 PRINT#s:NEXT
220 CLOSEOUT
```

Aufbau der Tabellen:

1. Spalte: Notenbezeichnung, wobei die Halbtöne als Erhöhung der darunterliegenden ganzen Note (mit #) angegeben sind.

2. Spalte: Frequenz der Note in Hertz (Schwingungen pro Sekunde). Alle Noten sind mit Kammerton A = 440 Hz als Bezugspunkt berechnet. Frequenzangaben für exakte Teilung, nicht temperiert o.ä.

3. Spalte: Die zugehörige Periodenlänge, mit der der PSG programmiert werden muß, ist sowohl auf drei Nachkommastellen genau, als auch ganzzahlig gerundet.

4. Spalte: Der relative Fehler, der bei der Rundung gemacht wurde, wird einmal in Prozent und einmal noch zusätzlich mit dem Abstand einer Prime (einem Halbtonschritt, Abstand zwischen zwei benachbarten Noten) gewichtet. Diese letztere Angabe ist dabei interessanter: 20 % hier heißen, daß sich die entsprechende Note durch die Rundung bereits um 20 % auf die benachbarte Note zubewegt hat.

***** Oktave -4 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	16.35	3822.256	3822	-0.0067	-0.1128
CIS	17.32	3607.730	3608	+0.0075	+0.1260
D	18.35	3405.243	3405	-0.0071	-0.1202
DIS	19.45	3214.122	3214	-0.0038	-0.0637
E	20.60	3033.727	3034	+0.0090	+0.1514
F	21.83	2863.457	2863	-0.0160	-0.2684
FIS	23.12	2702.743	2703	+0.0095	+0.1596
G	24.50	2551.050	2551	-0.0020	-0.0330
GIS	25.96	2407.871	2408	+0.0054	+0.0903
A	27.50	2272.727	2273	+0.0120	+0.2018
AIS	29.14	2145.169	2145	-0.0079	-0.1324
H	30.87	2024.770	2025	+0.0114	+0.1912

***** Oktave -3 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	32.70	1911.128	1911	-0.0067	-0.1128
CIS	34.65	1803.865	1804	+0.0075	+0.1260
D	36.71	1702.622	1703	+0.0222	+0.3737

DIS	38.89	1607.061	1607	-0.0038	-0.0637
E	41.20	1516.863	1517	+0.0090	+0.1514
F	43.65	1431.728	1432	+0.0190	+0.3189
FIS	46.25	1351.372	1351	-0.0275	-0.4626
G	49.00	1275.525	1276	+0.0372	+0.6262
GIS	51.91	1203.935	1204	+0.0054	+0.0903
A	55.00	1136.364	1136	-0.0320	-0.5381
AIS	58.27	1072.584	1073	+0.0387	+0.6516
H	61.74	1012.385	1012	-0.0380	-0.6394

***** Oktave -2 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	65.41	955.564	956	+0.0456	+0.7671
CIS	69.30	901.932	902	+0.0075	+0.1260
D	73.42	851.311	851	-0.0365	-0.6140
DIS	77.78	803.530	804	+0.0584	+0.9828
E	82.41	758.432	758	-0.0569	-0.9573
F	87.31	715.864	716	+0.0190	+0.3189
FIS	92.50	675.686	676	+0.0465	+0.7819
G	98.00	637.763	638	+0.0372	+0.6262
GIS	103.83	601.968	602	+0.0054	+0.0903
A	110.00	568.182	568	-0.0320	-0.5381
AIS	116.54	536.292	536	-0.0545	-0.9164
H	123.47	506.192	506	-0.0380	-0.6394

***** Oktave -1 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	130.81	477.782	478	+0.0456	+0.7671
CIS	138.59	450.966	451	+0.0075	+0.1260
D	146.83	425.655	426	+0.0810	+1.3614
DIS	155.56	401.765	402	+0.0584	+0.9828
E	164.81	379.216	379	-0.0569	-0.9573
F	174.61	357.932	358	+0.0190	+0.3189
FIS	185.00	337.843	338	+0.0465	+0.7819
G	196.00	318.881	319	+0.0372	+0.6262
GIS	207.65	300.984	301	+0.0054	+0.0903
A	220.00	284.091	284	-0.0320	-0.5381
AIS	233.08	268.146	268	-0.0545	-0.9164
H	246.94	253.096	253	-0.0380	-0.6394

***** Oktave +0 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	261.63	238.891	239	+0.0456	+0.7671
CIS	277.18	225.483	225	-0.2143	-3.6031
D	293.66	212.828	213	+0.0810	+1.3614
DIS	311.13	200.883	201	+0.0584	+0.9828
E	329.63	189.608	190	+0.2068	+3.4774
F	349.23	178.966	179	+0.0190	+0.3189
FIS	369.99	168.921	169	+0.0465	+0.7819
G	392.00	159.441	159	-0.2764	-4.6476
GIS	415.30	150.492	150	-0.3269	-5.4971
A	440.00	142.045	142	-0.0320	-0.5381
AIS	466.16	134.073	134	-0.0545	-0.9164
H	493.88	126.548	127	+0.3571	+6.0052

***** Oktave +1 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	523.25	119.446	119	-0.3730	-6.2725
CIS	554.37	112.742	113	+0.2292	+3.8552
D	587.33	106.414	106	-0.3889	-6.5404
DIS	622.25	100.441	100	-0.4394	-7.3889
E	659.26	94.804	95	+0.2068	+3.4774
F	698.46	89.483	89	-0.5398	-9.0779
FIS	739.99	84.461	84	-0.5455	-9.1737
G	783.99	79.720	80	+0.3508	+5.9000
GIS	830.61	75.246	75	-0.3269	-5.4971
A	880.00	71.023	71	-0.0320	-0.5381
AIS	932.33	67.037	67	-0.0545	-0.9164
H	987.77	63.274	63	-0.4331	-7.2840

***** Oktave +2 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	1046.50	59.723	60	+0.4642	+7.8068
CIS	1108.73	56.371	56	-0.6577	-11.0614
D	1174.66	53.207	53	-0.3889	-6.5404
DIS	1244.51	50.221	50	-0.4394	-7.3889
E	1318.51	47.402	47	-0.8480	-14.2615
F	1396.91	44.742	45	+0.5777	+9.7158
FIS	1479.98	42.230	42	-0.5455	-9.1737

G	1567.98	39.860	40	+0.3508	+5.9000
GIS	1661.22	37.623	38	+1.0021	+16.8525
A	1760.00	35.511	36	+1.3760	+23.1404
AIS	1864.66	33.518	34	+1.4372	+24.1702
H	1975.53	31.637	32	+1.1473	+19.2943

***** Oktave +3 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	2093.00	29.861	30	+0.4642	+7.8068
CIS	2217.46	28.185	28	-0.6577	-11.0614
D	2349.32	26.603	27	+1.4905	+25.0667
DIS	2489.02	25.110	25	-0.4394	-7.3889
E	2637.02	23.701	24	+1.2616	+21.2163
F	2793.83	22.371	22	-1.6573	-27.8716
FIS	2959.96	21.115	21	-0.5455	-9.1737
G	3135.96	19.930	20	+0.3508	+5.9000
GIS	3322.44	18.811	19	+1.0021	+16.8525
A	3520.00	17.756	18	+1.3760	+23.1404
AIS	3729.31	16.759	17	+1.4372	+24.1702
H	3951.07	15.819	16	+1.1473	+19.2943

***** Oktave +4 *****

Note	Frequenz (Hertz)	Periodenlänge genau	round	relativer Fehler in % 1/100	1/Halbton/100
C	4186.01	14.931	15	+0.4642	+7.8068
CIS	4434.92	14.093	14	-0.6577	-11.0614
D	4698.64	13.302	13	-2.2684	-38.1474
DIS	4978.03	12.555	13	+3.5431	+59.5842
E	5274.04	11.850	12	+1.2616	+21.2163
F	5587.65	11.185	11	-1.6573	-27.8716
FIS	5919.91	10.558	11	+4.1904	+70.4711
G	6271.93	9.965	10	+0.3508	+5.9000
GIS	6644.88	9.406	9	-4.3138	-72.5458
A	7040.00	8.878	9	+1.3760	+23.1404
AIS	7458.62	8.380	8	-4.5297	-76.1760
H	7902.13	7.909	8	+1.1473	+19.2943

Anhang H

Die Floppy

Portadressen zum Floppy-Controller

Ansteuerung der Laufwerkmotoren:

OUT &F8FF,0 – Stoppe Motor
OUT &F8FF,1 – Starte Motor

I/O-Adressen des FDC:

&FB7E – Haupt-Statusregister
&FB7F – Datenregister

Die Register des FDC

DAS HAUPT-STATUSREGISTER – INP(&FB7E)

Bit 7 – RQM – Request for Master
Bit 6 – DIO – Data Input/Output
Bit 5 – EXM – Execution Mode
Bit 4 – FCB – Floppy Controller Busy
Bits 0-3 – FDB – Floppy Drive Busy

DAS STATUSREGISTER 0

Bits 7, 6 – Interrupt Code
00 – Kommando erfolgreich beendet
01 – Kommando abgebrochen wegen Fehler
10 – ungültiges Kommando
11 – Kommando abgebrochen, weil Drive "not ready" wurde
Bit 5 = 1 – Seek-Ende auf einem Drive
Bit 4 = 1 – Fehler in der Floppy

Bit 3 = 1 – Laufwerk ist nicht bereit
Bit 2 – Head
Bit 1, 0 – Unit Select

DAS STATUSREGISTER 1

Bit 7 = 1 – End of track error
Bit 6 = 0 – nicht benutzt
Bit 5 = 1 – CRC-Fehler im Daten- oder ID-Feld
Bit 4 = 1 – Puffer-Überlauf
Bit 3 = 0 – nicht benutzt
Bit 2 = 1 – Sektor nicht auffindbar
Bit 1 = 1 – Diskette ist schreibgeschützt.
Bit 0 = 1 – ID- oder Data Address Mark fehlt

DAS STATUSREGISTER 2

Bit 7 = 0 – nicht benutzt
Bit 6 = 1 – "gelöschter" Sektor gefunden
Bit 5 = 1 – Checksummenfehler im Datenteil eines Sektors
Bit 4 = 1 – die logische Spurnummer aus der Sektor-ID stimmt nicht
Bit 3 = 1 – Vergleich von Sektor- und Prozessordaten lieferte Gleichheit
Bit 2 = 1 – Testbedingung im Scan-Kommando nicht erfüllt
Bit 1 = 1 – Track enthält fehlerhafte Stellen; nicht beschreiben
Bit 0 = 1 – Die Markierung für den Datenbereich war nicht auffindbar

DAS STATUSREGISTER 3

Bit 7 = 1 – Fehler-Flip-Flop gesetzt
Bit 6 = 1 – eingelegte Diskette ist schreibgeschützt
Bit 5 = 1 – Laufwerk meldet "ready"
Bit 4 = 1 – Schreib-Lesekopf steht auf Spur Null (Track 0)
Bit 3 = 1 – Doppelkopf-Laufwerk
Bit 2 – Head
Bit 1 – US1
Bit 0 – US0

Programmierung des FDC

OUT STANDARD-DATENBLOCK

```
OUT &FB7F,Spurnummer          ; Sektor-ID-Information
OUT &FB7F,Kopfnummer          ; dito
OUT &FB7F,Sektornummer        ; dito
OUT &FB7F,Sektorgröße         ; dito
OUT &FB7F,logische letzte Sektornummer der Spur
OUT &FB7F,Lücke zwischen Sektor-ID und Daten
OUT &FB7F,Sektorlänge wenn Sektorgröße = 0
```

INP STANDARD-DATENBLOCK

```
INP &FB7F,Statusregister 0
INP &FB7F,Statusregister 1
INP &FB7F,Statusregister 2
INP &FB7F,Spurnummer          ; Sektor-ID-Informationen
INP &FB7F,Kopfnummer          ; dito
INP &FB7F,Sektornummer        ; dito
INP &FB7F,Sektorgröße         ; dito
```

DIE BEFEHLE DES FDC

```
&X0ms00010 – ganze Spur lesen (READ TRACK)
&X00000011 – Laufwerkdaten festlegen (SPECIFY)
&X00000100 – Statusregister 3 abfragen (SENSE DRIVE STATE)
&Xtm000101 – Sektor(en) schreiben (WRITE SECTOR)
&Xtms00110 – Sektor(en) lesen (READ SECTOR)
&X00000111 – Spur 0 suchen (RECALIBRATE)
&X00001000 – Statusregister 0 abfragen (SENSE INTERRUPT STATE)
&Xtm001001 – gelöschte Sektoren schreiben (WRITE DELETED SECTOR)
&X0m001010 – Sektor-ID lesen (READ SECTOR ID)
&Xtms01100 – gelöschte Sektoren lesen (READ DELETED SECTOR)
&X0m001101 – eine Spur formatieren (FORMAT TRACK)
&X00001111 – Spur suchen (SEEK)
&Xtms10001 – Sektor(en) testen (SCAN EQUAL)
&Xtms11001 – Sektor(en) testen (SCAN LOW OR EQUAL)
&Xtms11101 – Sektor(en) testen (SCAN HIGH OR EQUAL)
```

Anhang I

BASIC

Die Token des Locomotive-BASIC

Um Platz zu sparen, aber auch, um die Programme schneller abarbeiten zu können, werden alle Schlüsselbegriffe von BASIC als Token abgespeichert. Dabei handelt es sich um Abkürzungen, die nur noch aus einem oder bei den Funktionen aus zwei Bytes bestehen. Aber auch einige häufig vorkommende Zeichen, wie etwa der Doppelpunkt oder kleine Zahlen, werden verkürzt dargestellt.

Token, die nur beim CPC 664 und 6128 vorhanden sind, sind mit ** markiert.

Byte	Bedeutung
00	Markierung für Zeilenende
01	Zeichen zum Statement-Trennen: :
02	Präfix für % -Variable
03	Präfix für $ -Variable
04	Präfix für ! -Variable
0B	Präfix für Integer-Variable ohne %, (Adresse bereits bestimmt)
0C	Präfix für String-Variable ohne $, (Adresse bereits bestimmt)
0D	Präfix für Real-Variable ohne !, (Adresse bereits bestimmt)
0D	Präfix für nicht gekennzeichnete Variable, deren Adresse noch nicht bestimmt ist
0E	Kürzel für die Zahl 0
0F	Kürzel für die Zahl 1
10	Kürzel für die Zahl 2
11	Kürzel für die Zahl 3
12	Kürzel für die Zahl 4
13	Kürzel für die Zahl 5
14	Kürzel für die Zahl 6
15	Kürzel für die Zahl 7
16	Kürzel für die Zahl 8
17	Kürzel für die Zahl 9

Byte	Bedeutung
19	Präfix für als Byte darstellbare Dezimalzahl
1A	Präfix für sonstige Integer-Dezimalzahl
1B	Präfix für binär angegebene Zahl
1C	Präfix für hexadezimal angegebene Zahl
1D	Präfix für Zeilenadresse
1E	Präfix für Zeilennummer
1F	Präfix für Fließkommazahl
20 bis 7F	ASCII-Zeichen (z.B. in Strings oder Variablennamen)

Ab &80 folgen die Token für die BASIC-Statements:

Byte	Bedeutung	Byte	Bedeutung
80	AFTER	9B	ERASE
81	AUTO	9C	ERROR
82	BORDER	9D	EVERY
84	CALL	9E	FOR
85	CAT	9F	GOSUB
86	CLEAR	A0	GOTO
87	CLG	A1	IF
88	CLOSEIN	A2	INK
89	CLOSEOUT	A3	INPUT
8A	CLS	A4	KEY
8B	CONT	A5	LET
8C	DATA	A6	LINE
8D	DEF	A7	LIST
8E	DEFINT	A8	LOAD
8F	DEFREAL	A9	LOCATE
90	DEFSTR	AA	MEMORY
91	DEG	AB	MERGE
92	DELETE	AC	MID$
93	DIM	AD	MODE
94	DRAW	AE	MOVE
95	DRAWR	AF	MOVER
96	EDIT	B0	NEXT
01,97	ELSE	B1	NEW
98	END	B2	ON
99	ENT	B3	ON BREAK
9A	ENV	B4	ON ERROR GOTO 0

Byte	Bedeutung	Byte		Bedeutung
B5	ON SQ	CD		SPEED
B6	OPENIN	CE		STOP
B7	OPENOUT	CF		SYMBOL
B8	ORIGIN	D0		TAG
B9	OUT	D1		TAGOFF
BA	PAPER	D2		T R O N
BB	PEN	D3		TROFF
BC	PLOT	D4		WAIT
BD	PLOTR	D5		WEND
BE	POKE	D6		WHILE
BF	PRINT	D7		WIDTH
01,C0	'	D8		WINDOW
C1	RAD	D9		ZONE
C2	RANDOMIZE	DA		WRITE
C3	READ	DB		D I
C4	RELEASE	DC		E I
C5	REM	DD	**	FILL
C6	RENUM	DE	**	GRAPHICS
C7	RESTORE	DF	**	MASK
C8	RESUME	E0	**	FRAME
C9	RETURN	E1	**	CURSOR
CA	RUN	E2		—
CB	SAVE			
CC	SOUND			

Ab &E3 folgen einige Token für reservierte Variablen, Operatoren und ähnliches.

Byte	Bedeutung	Byte	Bedeutung
E3	ERL	F4	+
E4	FN	F5	–
E5	SPC	F6	*
E6	STEP	F7	/
E7	SWAP	F8	↑
	---	F9	\
EA	TAB	FA	AND
EB	THEN	FB	MOD
EC	TO	FC	OR
ED	USING	FD	XOR
EE	>	FE	NOT

Byte	Bedeutung	Byte	Bedeutung
EF	=		
F0	>=	FF	Präfix für Funktionen
F1	<		
F2	<>		
F3	<=		

Mit &FF werden alle Funktionen gekennzeichnet. Um welche Funktion es sich handelt, wird durch das darauf folgende Byte bestimmt:

Byte	Bedeutung	Byte	Bedeutung
Funktionen mit einem Argument:		Funktionen ohne Argument:	
00	ABS	40	EOF
01	ASC	41	ERR
02	ATN	42	HIMEM
03	CHR$	43	INKEY$
04	CINT	44	PI
05	COS	45	RND
06	CREAL	46	TIME
07	EXP	47	XPOS
08	FIX	48	YPOS
09	FRE	49 **	DERR
0A	INKEY		
0B	INP	Funktionen mit mehreren Argumenten:	
0C	INT		
0D	JOY	71	BIN$
0E	LEN	72	DEC$
0F	LOG	73	HEX$
10	LOG10	74	INSTR
11	LOWER$	75	LEFT$
12	PEEK	76	MAX
13	REMAIN	77	MIN
14	SGN	78	POS
15	SIN	79	RIGHT$
16	SPACE$	7A	ROUND
17	SQ	7B	STRING$
18	SQR	7C	TEST
19	STR$	7D	TESTR
1A	TAN	7E **	COPYCHR$

Byte	Bedeutung	Byte	Bedeutung
1B	UNT	7F	VPOS
1C	UPPER$		
1D	VAL		

Priorität der Operationen in arithmetischen Ausdrücken in BASIC

I)	1.	Potenzierung	↑
	2.	Vorzeichenwechsel	–
	3.	Punktrechnung	* und /
	4.	Integer-Division	\
	5.	Restbildung	MOD
	6.	Strichrechnung	+ und –
II)	7.	Vergleich	< > <= >= <> und =
III)	8.	Komplement	NOT
	9.	Und	AND
	10.	Oder	OR
	11.	Exklusiv-Oder	XOR

Stichwortverzeichnis

Abbruchkriterium 84
Adressen (I/O) 242, 638
Adressierungsarten 107
AFTER 417
Amplituden-Hüllkurve 408
AMSDOS 367, 381, 465, 527
Anschlüsse 215, 642
Anschluß-Belegung der ICs 646
Argumente 94
Arithmetik-Vektoren 445
Array 33, 35
Audio-Anschluß 216
Ausführungszeiten Z80-Befehle 628

Bank 236
Bankswitching 115
BASIC 260
BASIC-Jumpblock 321, 445, 476, 582
BASIC-Programmzeile 261
BASIC-Token 687
BASIC-Stack 276
BASIC-Variablenbereich 269
BASIC-Zeiger 259
BCD 53
(packed) 53
Befehlselemente 89
Befehlssatz des Z80 615
Betriebssystem 307
Bildschirm-Modus 168
Bit 110
Black Box 65
Block 236, 390
Byte 53, 109

CALL 292
CASSETTE MANAGER 367, 371, 465, 527
Centronics-Anschluß 220
Chain 33, 37
Charakteristik 57
Circle 353, 448
CLUT 359
Codierung der Tinten (Inks) 665
Command 90
Controlbus 105
Controlcodes 332, 659
CPU 101
CRTC 142
Register 153, 662
Cursor 339

Daisy Chain 248
Datenkapselung 79
Datenspeicherung 33
Datenstrukturen 33
Dezimal-Wandlung 595
Disketten-Formate 388
Disketten-Inhaltsverzeichnis 389
Disketten-Monitor 395
Disketten-Motor 178
Disketten-Organisation 387
Disketten-Laufwerkanschluß 230
Drucker (8. Bit) 597
Drucker-Anschluß 220
Drucker-Spooler 438

Editor 451, 582
Eprom 208, 641
Programmierung 211
Timing-Diagramme 214

Erweiterungs-ROM 283
Erweiterungszeichen 657
Eventblock 428
Events 428
  synchrone 432
EVERY 417
Expansion-Port 224
Exponent 56, 57
Extent 389

Fakultät 85
Farbbyte 364
Farben 166, 359, 662
Farbnummern 166, 662
Fast Ticker 426
FDC 170
  Programmierung 179, 186, 685
  Haupt-Statusregister 180, 683
  Statusregister (0, 1, 2, 3) 182, 683
Fehlermeldungen 297
Fenster 338
Field 33, 95
Fill 85, 347
Firmware 455
Flags (Wirkung der Z80-Befehle) 624
Fließkomma-Routinen 477, 583
FLOATING POINT PACK 477
FLR 34
Force Legal 339
Frame Flyback 426
Frequenz-Hüllkurve 413
Funktionen 94, 587

Garbage Collection 61
Gliederung 65
GRAFIK VDU 340, 460, 502

Hardware-Erweiterungen 242
Header Record (cass) 372
Header Record (disc) 381
HIGH KERNEL JUMPBLOCK 320, 473, 569
HIMEM 257, 291

Hintergrund-Routinen 285
Hintergrund-ROM 252, 283
Hüllkurven
  Amplituden 408
  Frequenz 414
  PSG 139, 411, 676

I/O-Adressen 242, 638
Icon 334
ID 197
Illegals 628
Index 95
Indirection 322, 471, 563
Ink 166, 662
  Codierung im RAM 361, 665
  encoded 364, 666
Integer 55
Integer-Routinen 478, 591
Interrupt 87
  abstellen 436
  BASIC 417
  externer 117, 317, 434, 581
  in Maschinencode 425
  maskierbarer 117
  nicht maskierbarer 117
  normaler 117
  Sound 415
  Quellen 425, 434

Joystick-Anschluß 217
JUMPER 563

Kassettenrekorder-Anschluß 228
KEY MANAGER 323, 455, 480
KERNEL 468, 542
Klammern 94
komplementär 55, 56
Kommando (externes) 285
Konvertierungs-Routinen 593
Koordinaten 340
Kreise 353, 448

Laufindex 74
LCAT 314

Lightpen 142
Linien-Algorithmus 341
Linienmaske 347
Liste 41
LOW KERNEL JUMPBLOCK 474, 573
LOW LEVEL CASS 373
LOW LEVEL DISC 391

MACHINE PACK 470, 557
MAIN FIRMWARE JUMPBLOCK 321, 455
Mantisse 56, 57
Maschinencode 281, 289, 291
Merge-Fehler 382
Mikroprozessor 101
Modul 65, 79
Monitor-Anschluß 215

Nibble 110
Notenlänge 677

Operand 95
Operatoren 95, 585
Operatoren-Priorität 691
OS 307

PAL 158
Register 164, 641
Palettenfarbnummern 166
Parser 91
Parameter-Übergabe 293
Patch 312
Periodenlänge 406, 677
Pflichtenliste 65
Pictogramm-Vektor 334
Pin Out der ICs 646
PIO 121, 639
Pixel-Masken 363, 665
Plausibilitätskontrolle 71
Pollen 432
Precompilieren 265
PSG 131, 675
Prioritätshierarchie 96, 691
Programmbibliotheken 65
Programmiertechnik 65
Programmstrukturen 66

Queue 33, 46

RAM (externes) 599
RAM (IC dynamisches) 203
RAM-Konfiguration 165, 169, 237
RAM-Aufteilung 249, 254
RAM-Zeilen 356
RAM-Zuteilung (dynamische) 252
Real 56
Realzahl-Zerlegung 84
Record 34, 390
Refresh 204
Refresh-Register 111
Register 103, 632
Registersatz (zweiter) 112, 240
Rekursion 84
Relocalisator, Relocater 300
Reset 244
Reset-Schalter 601
Restart 6 (User) 319, 580
Restart-Vektoren 116, 241, 250, 309, 474, 573
Ringspeicher 48
ROM (externes) 246
ROM-Konfiguration 238
ROM-Selektion 239
ROM-Software 282
ROM-Status 168, 239
ROM-Typ 283
RSX 285, 292
RSX-Kommandos (AMSDOS) 385, 391
RSX-Lader 298

Schleife 72
FOR NEXT 73
abweisend 75
Schnittstellen 215
SCREEN PACK 354, 462, 514
scrollen (Hardware) 356
Sektor 390
Sektor-Read/Write 396

Semaphoren 441
Semantik 98
Soft End 382
Separator 93
SOUND-MANAGER 401, 467, 537
Sound-Statement 402
Speicheraufteilung 249, 254, 637
Speicherkonfiguration 235
Spooler 438
Sprung 67
  bedingter 68
  indirekter 70
  unbedingter 67
Sprungleiste 307, 317
Stack 33, 42
Statement 91
Steuerzeichen 658
String-Descriptor 60
Stromversorgung 215
Synchronisations-Zeichen 374
Syntax 98
Systembus-Anschluß 224
Systemvariablen
  AMSDOS 610
  BASIC 608
  Betriebssystem 603

Tastatur 323, 651
  Schaubilder 651
Tastennummern 651
Tasten-Belegung 655
Term 98
TEXT VDU 332, 457, 488
Ticker 427
Timing-Diagramme Z80 633
Tonperiodenlänge 677
Token 263, 687
Tree 33, 40

ULA 158
  Register 164, 640, 664
Universal-Ticker 427
Unterprogramm 76
User Restart 319

Variablen 79
  globale 79
  lokale 79
Vektor 309
Vektordrehung 352
Verzweigung 67
Video-RAM 354
Vielfarben-Zeichen 365
VLR 34
Vordergrund-Programm 281

Word 55, 109

Z80 101
  Befehlsausführungszeiten 628
  Befehlssatz 615
  Illegals 628
  Register 103, 632
  Timing-Diagramme 633
Zeichensatz 337, 667
Zeichenmatrix 337
Zufallsgenerator 584



von Reinhold Krumscheid

Das Textverarbeitungs-Programm der Spitzenklasse auch für den Schneider CPC 464/664/6128. Mit Profi-Möglichkeiten zum kleinen Preis.

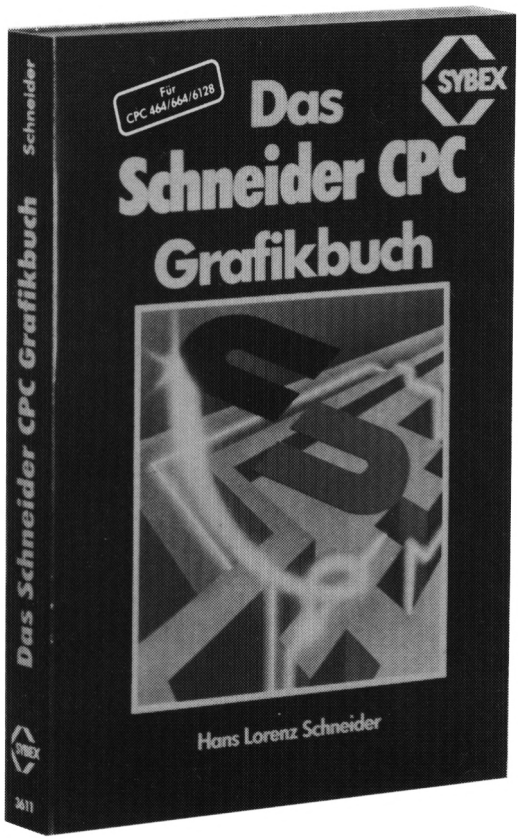
Diskette + Trainingsbuch, Best.-Nr. **3416** (1986)



Reinhold Krumscheid

Autor des CPC StarTexter, hat hier eine leistungsstarke Dateiverwaltung – kompatibel zur Textverarbeitung – vorgelegt; anwendbar auf CPC 464, 664 und 6128. Diskette mit ausführlichem Trainingsbuch.

Best.-Nr. **3423** (1986)



von H. L. Schneider

führt Sie schrittweise in die vielfältigen Grafikmöglichkeiten Ihres CPC 464, 664 oder 6128 ein; vom Grundlagenwissen bis zur Hardcopy.

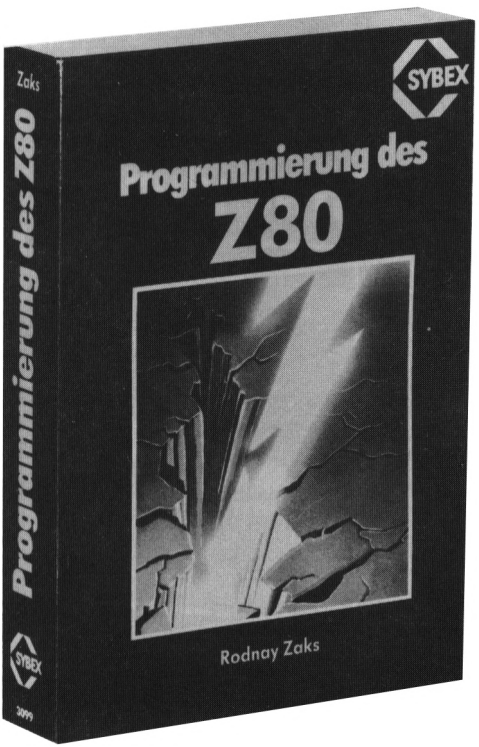
328 Seiten, zahlr. Abbildungen, Best.-Nr. **3611** (1986)



von Karl-Hermann Rollke

Der Sybex-Bestseller „Arbeiten mit Turbo Pascal" liegt jetzt in der überarbeiteten Version für alle Schneider CPC-Benutzer vor. Enthalten sind zusätzliche Informationen, zugeschnitten auf die für den Schneider CPC gültige Arbeits- und Systemumgebung.

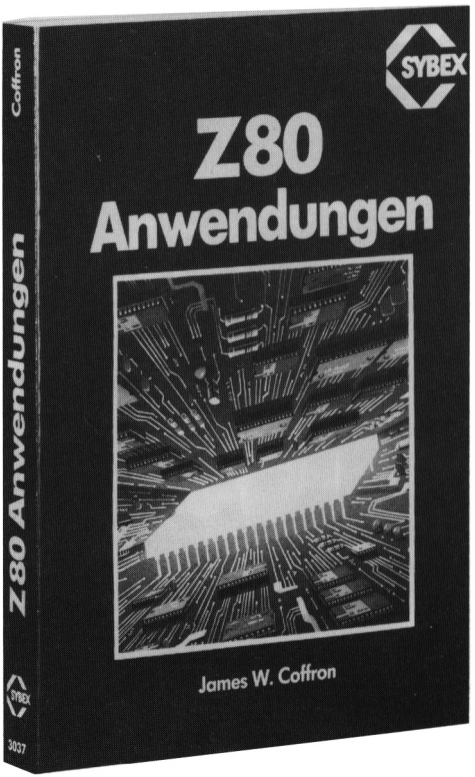
296 Seiten, mit Abb., Best.-Nr. **3649** (1986)



von Rodnay Zaks

ein umfassendes Nachschlagewerk zum Z80-Mikroprozessor – jetzt in einer durch Lösungen ergänzten Ausgabe. 2., erweiterte Ausgabe.

640 Seiten, 176 Abbildungen, Best.-Nr.: **3099** (1985)



von J. W. Coffron

vermittelt alle nötigen Anweisungen, um Peripherie-Bausteine mit dem Z80 zu steuern und individuelle Hardware-Lösungen zu realisieren.

296 Seiten, 204 Abbildungen, Best.-Nr.: **3037** (1984)

Die SYBEX-Bibliothek

Schneider

SCHNEIDER CPC 464: MEIN ERSTES BASIC PROGRAMM
von Rodnay Zaks – zahlreiche farbige Illustrationen und viele Diagramme helfen Ihnen, auf spielerische Weise in BASIC zu programmieren; ohne Vorkenntnisse nutzbar. 208 Seiten, zahl. farb. Abb., Best.-Nr. **3096** (1985)

MEIN SCHNEIDER CPC
von Norbert und Christoph Hesselmann – von der Hardware-Umgebung über ein umfangreiches BASIC-Lexikon, Grafikentwurf und Musikerzeugung, Mikroprozessorentechnik und Maschinensprache, Arbeiten mit dem Disketten-Laufwerk bis hin zu den Betriebssystemen AMSDOS und CP/M 2.2. 376 Seiten, 124 Abbildungen, Best.-Nr.: **3602** (1985)

ARBEITEN MIT DEM SCHNEIDER CPC
von Hans Lorenz Schneider – eine umfassende und didaktisch aufbereitete Arbeitshilfe für Anfänger, aber auch Fortgeschrittene finden ein Bündel von Tips und Tricks. 288 Seiten, 113 Abbildungen, Best.-Nr.: **3603** (1985)

SCHNEIDER CPC 464 ASSEMBLER KURS
– Das Buch führt Sie schrittweise in die Programmierung des Z80 ein und vermittelt Ihnen Befehlssatz des Prozessors wie Adressierungsarten. 232 Seiten, mit Abbildungen, Buch und Kassette, Best.-Nr.: **3412** (1985), Buch und Diskette, Best.-Nr.: **3418** (1986)

SCHNEIDER CPC – ARBEITEN CP/M
von Wolfgang Höfs führt in leicht verständlicher Form in die Betriebssystem-Umgebung CP/M 2.2 für die Systeme 464/664 sowie CP/M 3.0 für den CPC 6128 ein und erläutert den Umgang mit den Kommandos sowie den Dienstprogrammen. Ca. 250 Seiten, ca. 45 Abb., Best.-Nr. **3637** (1987)

SCHNEIDER CPC – ERFOLG MIT MULTIPLAN
von Thorsten Ritter – Besitzer eines CPC 6128 werden sich schnell mit dieser überarbeiteten und auf ihre System-Umgebung zugeschnittenen Version des Originals „Erfolg mit Multiplan" anfreunden und als didaktisch gut aufbereitetes Lehr- und Nachschlagwerk für die Tabellenkalkulation schätzen lernen. 200 Seiten, ca. 45 Abb., Best.-Nr. **3639** (1986)

SCHNEIDER CPC – EINFÜHRUNG IN WORDSTAR
von Arthur Naiman – Der SYBEX-Bestseller „Einführung in WordStar" in einer auf den Schneider CPC zugeschnittenen Version – ergänzt durch wertvolle Hinweise für die Installation von Druckern sowie Systempatches. 280 Seiten, ca. 40 Abb., Best.-Nr. **3646** (1986)

SCHNEIDER CPC – ARBEITEN MIT dBASE II
von Michael A. Beisecker – Nutzer eines Schneider CPC 6128 oder 8256/512 bekommen alle notwendigen Kenntnisse für den erfolgreichen Einsatz des populären Datenbank-Programms dBASE II. Die Lernschritte mit wachsendem Schwierigkeitsgrad werden durch praxisgerechte Beispiele ergänzt: Installation von und Programmieren mit dBASE II, Tips und Tricks, Editieren von Befehlsdateien mit WordStar u.v.m. 272 Seiten, mit Abb., Best.-Nr. **3660** (1986)

Einführende Literatur

SYBEX MIKROCOMPUTER LEXIKON
– die schnelle Informationsbörse! Über 1500 Definitionen, Kurzformeln, Begriffsschema der Mikroprozessor-Technik, englisch/deutsches und französisch/deutsches Wörterbuch, Bezugsquellen. 192 Seiten, Format 12,5 x 18 cm, Best.-Nr.: **3035** (1984)

COMPUTER TOTAL VERRÜCKT
von Daniel Le Noury – mit diesem Buch kommen Sie wieder zur Besinnung, nachdem Sie sich halbtot gelacht haben. Ca. 100 Cartoons rund um den Computer. 96 Seiten, Best.-Nr. **3042** (1984)

Sprachen

BASIC

BASIC PROGRAMME – MATHEMATIK, STATISTIK, INFORMATIK
von Alan Miller – eine Bibliothek von Programmen zu den wichtigsten Problemlösungen mit numerischen Verfahren, alle in BASIC geschrieben, mit Musterlauf und Programmlisting. 352 Seiten, 147 Abbildungen, Best.-Nr.: **3015** (1983)

GRUNDKURS IN BASIC
von U. Ströbel – die Einführung in die meistgenutzte Programmiersprache für Lehrer, Schüler und das Selbststudium (Reihe SYBEX Informatik). 208 Seiten, mit Abb., Best.-Nr. **3058** (1985), Lehrerbegleitheft Best.-Nr. **3091** (1985)

Pascal

DAS PASCAL HANDBUCH
von Jacques Tiberghien – ein Wörterbuch mit jeder Pascal-Anweisung und jedem Symbol, reservierten Wort, Bezeichner und Operator, für beinahe alle bekannten Pascal-Versionen incl. Turbo Pascal. 520 Seiten, 270 Abbildungen, Format 23 x 18 cm, Best.-Nr.: **3614** (1986)

DAS ARBEITSBUCH ZU TURBO PASCAL
von Karl-Udo Bromm u.a. – Der Autor, ein erfahrener Pädagoge, vermittelt dem mit Turbo Pascal arbeitenden Leser eine Fülle wertvoller Routinen zu den unterschiedlichsten Themenbereichen für Schule, Hobby und Beruf. Ca. 320 Seiten, ca. 60 Abb., Best.-Nr. **3629** (1987)

Fordern Sie ein Gesamtverzeichnis unserer Verlagsproduktion an:

SYBEX-VERLAG GmbH
Vogelsanger Weg 111
4000 Düsseldorf 30
Tel.: (02 11) 61 80 2-0
Telex: 8 588 163

SYBEX INC.
2021 Challenger drive, NBR 100
Alameda, CA 94501, USA
Tel.: (4 15) 523-8233
Telex: 287 639 SYBEX UR

SYBEX
6–8, Impasse du Curé
75018 Paris
Tel.: 1/203-95-95
Telex: 211.801 f

SYBEX

Das Schneider CPC Systembuch

Auch wenn Ihnen Ihr Computer einmal abhanden kommen sollte: Dieses Buch dürfen Sie nicht aus der Hand geben!

Die *Tips & Tricks*, das *Gewußt wie* und die vielen anschaulichen *Beispiele* machen es zu einem Werk, das auch Anfängern sehr schnell die „Höheren Weihen" der Maschinensprache zukommen läßt. Zusammen mit den umfassenden und schnell auffindbaren *Informationen* wird es aber gerade für den Profi zur unentbehrlichen Hilfe.

Aus dem Inhalt:

- Die Schneider CPC's für Unentschlossene
- Programmierstil, Programmelemente
- Die Feinheiten des Locomotive BASIC
- Überblick über Z80-Assembler
- Maschinensprache auf dem CPC
- Die Firmware: in- und auswendig
- Unterschiede und Gemeinsamkeiten der CPC's
- Ein Anhang, der es *in* sich hat

In verständlichem Stil werden die verschiedenen Abteilungen im Schneider CPC erklärt: was sie tun, wie sie benutzt werden und wie man noch mehr herausholen kann. Dabei wird mit Beispielen und Darstellungen nicht gegeizt. Die Grafiken, Tabellen und Verzeichnisse im Anhang machen dieses Buch zum ersten Nachschlagewerk bei allen Fragen, die beim Programmieren auftreten können.

Über den Autor:

Günter Woigk studiert Informatik und ist seit über vier Jahren „dabei". Nach dem geradezu klassischen Einstieg über TI-58 und TI-59 und Aufstieg mit einem ZX-Spectrum ist bei ihm nun meist ein Schneider CPC im Einsatz. Nach einigen Veröffentlichungen in Zeitschriften ist dies sein erstes Buch.

ISBN 3-88745-606-8

DM 58,—
sFr 53,40
öS 452,—



3606

SYBEX

Das Schneider CPC Systembuch

Woigk

AMSTRAD CPC



MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.