Inhaltsverzeichnis		Seite 1
Danksagung / Hinweis		Seite 4
Teil 1	Ausgabe 03/92	Seite 5
Grundlagen der Datenein- / Ausgabe		Seite 5
Floppybefehlskanal		Seite 7
NEW-Befehl		Seite 7
Rename-Befehl		Seite 8
Validate-Befehl		Seite 8
Scratch-Befehl		Seite 8
Die verschiedenen Dateitypen		Seite 8
Programmierung sequentieller Dateien		Seite 9
Teil 2	Ausgabe 06/92	Seite 11
Relative Datenverwaltung		Seite 11
Die Programmierung		Seite 12
Der Schreibzugriff		Seite 13
Der Lesezugriff		Seite 14
Databanking		Seite 15
Datenfelder		Seite 15
Sortieren und Indizieren		Seite 17
Teil 3	Ausgabe 07 & 08/92	Seite 19
Diskettenstruktur		Seite 19
Die Floppy – ein eigenständiger Computer		Seite 20
Direktzugriffsbefehle		Seite 21
Blockbefehle		Seite 21
Memorybefehle		Seite 23
Userbefehle		Seite 24
Teil 4	Ausgabe 09/92	Seite 25
Was steht wo?		Seite 25
Die Datenblöcke		Seite 26
Directory-Blöcke		Seite 26
Disk-Header-Block		Seite 27
BAM (Block Availability Map)		Seite 28
Directory-Block		Seite 29
Teil 5	Ausgabe 10/92	Seite 31
Diskette Schreibschützen		Seite 31
Ändern des Diskettennamens und der ID		Seite 33
Teil 6	Ausgabe 11/92	Seite 35

Directory anzeigen		Seite 35
Gelöschte Files retten		Seite 37
Teil 7	Ausgabe 12/92	Seite 42
Einstieg in Assembler		Seite 43
Öffnen und Schliessen von Filesl		Seite 43
Der Datenaustausch		Seite 44
Fehlererkennung und -behandlung		Seite 45
Zusammenfassung		Seite 46
LOAD und SAVE		Seite 48
Teil 8	Ausgabe 01/93	Seite 49
Die alte Methode		Seite 49
Die neue Methode		Seite 50
Programmbeispiele		Seite 52
1. Lesen und Schreiben eines Files		Seite 52
2. Floppystatus anzeigen		Seite 53
3. BAM anzeigen		Seite 53
Teil 9	Ausgabe 02/93	Seite 56
Die Aufgabe		Seite 57
Theorie		Seite 57
Die benötigten Programmteile		Seite 58
Benutzte Speicherstellen		Seite 58
Steuerroutinen		Seite 59
IO-Routinen		Seite 59
Praxis		Seite 60
Main-Routine		Seite 60
STRIEC-Routine		Seite 63
SENDCOM-Routine		Seite 63
WDUMMY-Routine		Seite 64
GETNEED-Routine		Seite 64
GETDISKF-Routine		Seite 65
GETDIRF-Routine		Seite 66
CHGENT-Routine		Seite 66
WBLOCKS-Routine		Seite 68
Programmlistings		Seite 70
FK.NAMECHANGE	Ausgabe 10/92	Seite 70
FK.SCHREIBSCHUTZ		Seite 71
FK.UNDEL	Ausgabe 11/92	Seite 71

FK.DIR		Seite 74
FK.IO.S	Ausgabe 01/93	Seite 74
FK.SHOWBAM.S		Seite 76
FK.USEDIR.SRC	Ausgabe 03/93	Seite 80
Steuerzeichenerklärung		Seite 91

Danksagung

Die Veröffentlichung der Magic Disk Kurse als PDF-Datei erfolgt mit schriftlicher Genehmigung der COMPUTEC MEDIA GmbH. Es dürfen keine kommerziellen Absichten verfolgt werden!

Mein Dank gilt ebenfalls Mirco Geldermann. Auf seiner Webseite www.magicdisk64.de ist es möglich die Diskettenimages der Magic Disk C64 Ausgaben offiziell herunterzuladen.

Teil 1 – Magic Disk 03/92

In Anlehnung an diverse Geräusche die das Laufwerk unseres Computerlieblings an und wann von sich gibt, wünsche ich Sie herzlich willkommen zum ersten Teil unseres neuen Floppy-kurses.
In den nächsten Monaten will ich Ihnen hier die Funktionsweise und effektive Benutzung des Floppylaufwerks 1541 des 64ers erklären. Beginnend mit den Grundlagen zur Datenspeicherung auf Diskette werden wir uns über sequentielle und relative Dateiverwaltung immer mehr an die 'Innereien' der Floppy herantasten, und uns auch mit dem Diskettenaufbau und Direktzugriff beschäftigen. Danach will ich Ihnen zeigen, wie einfach es ist die Floppy in Assembler zu programmieren und Ihnen desweiteren meine gesammelten Tips und Tricks zur Floppy offenbaren. Ich wünsche Ihnen also viel Spaß bei diesem Kurs und will gleich zu Sache kommen...

1. GRUNDLAGEN DER DATEN EIN- / AUSGABE

Dieser erste Teil des Floppy-Kurses soll ganz den Grundlagen und der sequentiellen File-programmierung gewidmet sein.
Kommen wir zunächst einmal zu den Grundbegriffen der Daten Ein- und Ausgabe des 64ers. Wenn Sie sich mit BASIC auskennen, so wissen Sie, daß es prinzipiell 5 BASIC-Befehle gibt, mit denen wir Daten an externe, oder von externen Geräten senden und empfangen können. Diese sind:

- OPEN zum Öffnen eines Datenkanals,
- PRINT# zum Senden von Daten in einen Datenkanal,
- GET# zum Lesen von Daten von einem Datenkanal,
- INPUT# zum einfacheren (BASICorientierten) Lesen von einem Datenkanal, und
- CLOSE zum abschließenden Schließen eines Datenkanals.

Daten Ein- und Ausgaben werden nun prinzipiell mit drei verschiedenen Parametern zu den obigen Befehlen gesteuert.
Sie geben an, welchem logischen Kanal der aktuelle Datenaustausch zugeordnet wird und geben Informationen über die Art des Zugriffs (lesen/ schreiben) und über das entsprechende Gerät, das angesprochen werden soll. Diese drei Parameter sind die "logische Filenummer", die "Sekundäradresse" und die "Gerätenummer".
Die logische Filenummer ist eine Zahl zwischen 1 und 127, die als Kennung für eine Ein-/ Ausgabeoperation dient. Durch den OPEN-Befehl wird ihr eine bestimmte Fileoperation zugewiesen, die sie eindeutig definiert. Wird also ein Datenkanal zur Floppy mit der logischen Filenummer 1 und ein Datenkanal zum Drucker mit der logischen Filenummer 2 geöffnet, so weiß das Betriebssystem des 64ers immer, wohin es Daten mit einer der beiden Filenummern senden soll. Ein "PRINT#1,A$" sendet so die Stringvariable A$ an den momentan offenen Floppykanal (nachdem die Filenummer 1 so durch 'OPEN' definiert wurde), wohingegen der Befehl "PRINT#2,A$" dieselbe Stringvariable an den Drucker sendet (nachdem ihm die Filenummer 2 zugeordnet wurde).
Die Gerätenummer spezifiziert nun das Gerät, mit dem Daten ausgetauscht werden sollen. Sie kann Werte zwischen 0 und 15 annehmen. Hierzu gibt es eine Liste, die die verschiedenen Geräte spezifiziert.
Das Betriebssystem des C64 benötigt diese Nummer deshalb, weil die verschiedenen Peripheriegeräte mit unterschiedlichen Betriebssystemroutinen angesprochen werden müssen.

Die Verteilung der Gerätenummern an die entsprechenden Geräte ist wie folgt festgelegt:

Gerätenummer	Gerät
0	Bildschirm
1	Datasette
2	RS232-Schnittstelle
4	Drucker 1
5	Drucker 2
8	Floppylaufwerk 1
9	Floppylaufwerk 2
11	Floppylaufwerk 3
10	Floppylaufwerk 4

Die restlichen Nummern sind unbelegt.

Wie Sie sehen ist es also möglich bis zu 4 Floppylaufwerke und 2 Drucker an den C64 anzuschließen, die alle getrennt voneinander angesprochen werden können. Ebenso ist die Kommunikation mit dem Bildschirm über die Gerätenummer 0 möglich. Hierbei empfangen Sie direkt die Eingaben, die gerade vom Benutzer auf den Bildschirm geschrieben werden.
Die Sekundäradresse einer Ein-/ Ausgabeoperation gibt nun die Art einer Datenoperation an. Sie kann Werte zwischen 0 und 15 annehmen, wobei es folgende Bedeutungen gibt:

Sekundäradresse	Bedeutung
0	Programm laden
1	Programm speichern
2 – 14	Daten Lesen oder Schreiben
15	Floppybefehlskanal

In der Regel werden die Sekundäradressen zwischen 2 und 14 verwendet. Die Adressen 0,1 und 15 sind spezielle Adressen, die eigens für die Verwaltung von Massenspeichern gedacht sind. Normalerweise muß man nämlich beim Zugriff auf einen Massenspeicher (so wie auch das Floppylaufwerk einer ist) grundsätzlich die Art des Zugriffs explizit im Filenamen angeben. Möchte man aber ein Programmfile (dazu später mehr) lesen oder schreiben, so kann man sich die Angabe im Filenamen sparen und die Adressen 0 oder 1 benutzen. Die Floppy weiß in dem Fall direkt, daß sie ein Programmfile lesen, bzw. schreiben soll. Dazu will ich Ihnen später noch ein paar Beispiele liefern.
Die Sekundäradresse 15 ist ausschließlich für das Floppylaufwerk reserviert. Mit ihr wird der sogenannte Floppybefehlskanal geöffnet. Dieser ist nicht an ein spezielles File gebunden und dient der Übertragung von Befehlen an die Floppy selbst. Die 1541 verfügt nämlich, wie der C64 auch, über einen eigenen Mikroprozessor (den 6502, ein Artverwandter des 6510, der im 64 er seinen Dienst verrichtet) und eigene Ein-/ Ausgabechips. Sie stellt im Prinzip einen autonomen Computer dar, der richtig Programmiert werden kann. Die wichtigsten Funktionen sind in speziellen Floppybefehlen zusammengefasst und werden über den erwähnten Befehlskanal aufgerufen (wie z.B. der Befehl, die einliegende Diskette zu formatieren, oder ein spezielles File von der einliegenden Diskette zu löschen). Der Floppybefehlskanal hat später, wenn wir die Direktzugriffsbefehle behandeln, und in Zusammenhang mit der relativen Dateiverwaltung eine große Bedeutung. Die obig beschriebenen Parameter, die der OPEN-Befehl benötigt müssen wie folgt angewandt werden (Praxisbeispiele werden wir im Laufe dieses Kurses noch genügend kennenlernen, deshalb hier nur eine Syntaxdefinition):

OPEN (**logische. Filenumme**r) , (**Gerätenummer**) , (**Sekundäradresse**)

DER FLOPPYBEFEHLSKANAL

Wir wollen uns nun mit der Benutzung des Floppybefehlskanals beschäftigen und einmal die alltäglichen Floppybefehle durchgehen.
Bevor Sie also einen Befehl an die Floppy schicken, müssen Sie den Befehlskanal öffnen. Dies geschieht mit dem Befehl "OPEN 1,8,15". Wir öffnen hier einen Kanal mit der logischen Filenummer 1, verbunden mit dem Gerät Nummer 8 (der Floppy nämlich) und mit der Sekundäradresse 15 (eben dem Befehlskanal derselbigen). Nun können Sie der Floppy Befehle senden, die diese dann unabhängig vom 64 er bearbeiten wird. Sie können also Ihren 64 er währenddessen in seinem aktuellen Programm fortfahren lassen. Er arbeitet, solange die Floppy selbst arbeitet, unabhängig von ihr. Nur, wenn während dieser Zeit ein weiterer Diskettenzugriff notwendig wird, wird der 64er angehalten (s.u.). Ich will Ihnen nun die Standard-Floppybefehle auflisten, die wir einfach benutzen können. Später beim Direktzugriff und bei der relativen Dateiverwaltung werden wir ebenfalls über den Befehlskanal der Floppy Anweisungen geben, uns die Daten, die wir von ihr verlangen entsprechend vorzubereiten.

Kommen wir jedoch erst einmal zu den einfachen Befehlen:

Der NEW-Befehl:

Mit diesem Befehl formatieren wir eine neue Diskette. Grundsätzlich bestehen die Floppybefehle aus einem oder mehreren Buchstaben, sowie den zu jedem Befehl variierenden Parametern. Beim NEW-Befehl ist die Befehlskennung ein "**N:**", nach diesen beiden Zeichen folgt nun der Name, den die Diskette erhalten soll, sowie eine zwei Zeichen lange Identifikationskennung ("ID"). Floppybefehle werden nach dem Öffnen des Befehlskanals immer mit einem "PRINT# lfn" (lfn = logische Filenummer) an die Floppy geschickt. Mit dem folgenden BASIC-Kommando fordern wir die Floppy also dazu auf, die einliegende Diskette mit dem Namen "MEINE DISK" und der ID "MD" zu formatieren. Der Befehlskanal wurde unter der logischen Filenummer 1 geöffnet (wie in obigem Beispiel), also senden wir den Befehl auch an Kanal 1:

```
PRINT#1,"N:MEINE DISK,MD"
```

Die Floppy beginnt nun mit der Formatierung der Diskette. Der 64er meldet sich mit einem "READY." zurück, und Sie können während der Formatierung mit ihm arbeiten. Sie können den Befehlskanal nun offen halten und weitere Floppybefehle senden, oder aber auch irgend eine andere Tätigkeit mit dem Rechner tun. Achten Sie allerdings darauf, daß beim weiteren Senden eines Befehls, sowie dem Schließen des Befehlskanals, der Rechner blockiert wird. Das liegt daran, daß die Floppy während einer internen Operation dem 64 er signalisiert, daß sie gerade "BUSY", also am Arbeiten ist. Soll der C64 nun mit der Floppy kommunizieren, so wartet er solange, bis die Floppy wieder frei wird. Demnach führt jeder weitere Befehl, der die Floppy anspricht unweigerlich zu einem zwischenzeitlichen Stop des Rechners.
Der NEW-Befehl kennt übrigens zwei Syntaxen. Übergeben Sie einen Namen UND eine ID, dann wird die Diskette physisch formatiert. Das heißt, sie wird Spur um Spur neu angelegt, wobei die neuen Spuren mit Nullen aufgefüllt werden. Daten, die sich evtl. auf ihr befanden werden somit komplett gelöscht. Übergeben Sie aber nur einen Diskettennamen, und keine ID, so wird die Diskette "softformatiert". Es wird lediglich der neue Name über den alten geschrieben, und alle Blocks als unbelegt gekennzeichnet. Die ID bleibt die alte. Die alten Daten sind jedoch immer noch auf den Spuren enthalten und können evtl. gerettet werden.
Ausserdem ist das softformatieren weitaus schneller als ein "hardformatieren". Es funktioniert allerdings nur bei schon einmal hardformatierten Disketten, da diese die grundlegende Diskettenstruktur schon enthalten.
Sie schließen den oben geöffneten Befehlskanal übrigens wieder mit "CLOSE 1". Tun Sie das bitte immer, wenn Sie ein Datenfile, oder den Befehlskanal nicht mehr brauchen, da der 64er intern immer nur 10 offene Kanäle verwalten kann.

Der RENAME-Befehl:
Mit dem Rename-Befehl können Sie den Namen eines schon bestehenden Files umbenennen. Die Syntax ist eigentlich recht einfach. Das Kürzel für RENAME ist "**R:**" es folgen nun der neue und der alte Filename getrennt durch ein "="-Zeichen.
Hier ein Beispiel:

```
PRINT#1,"R:MAGIC DISK=GAME ON".
```

Dieser Befehl benennt das File "GAME ON" in "MAGIC DISK" um. Wieder benutzten wir die logische Filenummer 1, die oben im OPEN-Befehl definiert wurde.

Der VALIDATE-Befehl:
VALIDATE bedeutet "überprüfen", und selbiges tut der Validate-Befehl der Floppy. Haben Sie nämlich Grund zu glauben, daß die Diskettenstruktur durcheinander gekommen ist, so zum Beispiel wenn Sie ein, oder mehrere Files speichern wollten, für die aber nicht ausreichend Platz vorhanden war, dann sollten Sie diesen Befehl verwenden. Er untersucht die Diskette auf ihre korrekte Struktur und korrigiert alles, was nicht einer normalen Diskettenstruktur entspricht. In dem Beispiel mit einem nur teilweise gespeicherten Programm sind die schon geschriebenen Blocks dieses Files als belegt gekennzeichnet, obwohl das File im Directoryeintrag mit "0 Blocks" und einem "*" als unbrauchbar gekennzeichnet ist. Der Validate-Befehl erkennt nun die fälschlicherweise belegten Blocks und gibt sie wieder frei. Desweiteren löscht er den markierten Eintrag aus dem Directory. Er benötigt keinerlei Parameter und kann folgendermaßen aufgerufen werden:

```
PRINT#1,"V"
```

Die Floppy beginnt nun mit der Validierung. Diese kann je nach dem wie voll und wieviele einzelne Files auf der Diskette enthalten sind, bis zu mehrere Minuten in Anspruch nehmen.

Der SCRATCH-Befehl:
Dieser Befehl löscht ein File auf Diskette. Englisch "to scratch" bedeutet "kratzen" und im Übertragenen Sinne "kratzt" die Floppy tatsächlich ein File von der Diskettenoberfläche.
Der Scratch-Befehl wird mit "**S:**" eingeleitet, gefolgt von dem oder den Filenamen, die gelöscht werden sollen. Hierbei dürfen Sie sogar sogenannte Filepatterns benutzen, die eine Fileangabe abkürzen.
Ein "Pattern" ist eine Maske, die man der Floppy für einen Filenamen übergibt.
Alle Files, deren Namen dieser Maske entsprechen sind damit angesprochen, werden in unserem Fall also durch den Scratch-Befehl gelöscht. Hier einige Beispiele:

```
PRINT#1,"S:FILE1"
PRINT#1,"S:TEST1,TEST2,TEST10,TEST11"
PRINT#1,"S:TEST?"
PRINT#1,"S:TEST*"
```

Im ersten Beispiel wird das File namens "FILE1" gelöscht. Das zweite Beispiel löscht die Files "TEST1","TEST2","TEST10", und "TEST11". Selbiges können wir aber auch mit Filepatterns verkürzen. So können wir zum Beispiel mit dem Fragezeichen eine Fileangabe abkürzen.
es steht für ein beliebiges Zeichen. Das dritte Beispiel löscht also gleichzeitig die Files "TEST1" und "TEST2". Im vierten Beispiel benutzen wir den Asterisk ("*") als Abkürzung. Er steht für alles, was hinter den Zeichen " TEST" folgt. Wir löschen mit diesem Befehl also auf Einmal alle vier Files aus dem zweiten Beispiel.

DIE VERSCHIEDENEN DATEITYPEN
Nachdem wir nun Grundsätzliches über die Kommunikation mit der Floppy gelernt haben, möchte ich Sie nun in die einzelnen File-Arten selbiger einführen. Das benötigen wir, um die unterschiedliche Programmierung der einzelnen File-Typen zu verstehen. Insgesamt kennt die

1541 fünf verschiedene File-Typen. Diese sind:

- Programm-Dateien (PRG):
  In diesen Files sind Daten sequentiell, also Byte hinter Byte abgespeichert. Sie enthalten ausschließlich Programmdaten, also direkt ausführbare BASIC-, oder Maschinenprogramme.
- Sequentielle Dateien (SEQ):
  Diese Fileart unterscheidet sich in nichts mit der vorherigen. Die Daten liegen hier ebenfalls sequentiell auf der Diskette vor. Der einzige Grund, warum man zwei Filearten zur sequentiellen Speicherung gewählt hat, ist der, daß in SEQ-Files immer nur Daten von und zu Programmen gespeichert werden sollen und keine Programme selbst. Wenn Sie also eine Adressverwaltung programmieren, so sollten Sie Ihre Adressen in einer SEQ-Datei speichern. Dies kennzeichnet Ihre Daten eben als Daten und nicht als ausführbares Programm (obwohl Sie genauso gut eine PRG-Datei verwenden könnten!). Die Unterscheidung ist besonders wichtig für den LOAD-Befehl, der SEQ-Dateien gar nicht erst liest, sondern ausschließlich PRG-Dateien in den Speicher des 64ers transferiert.
- Relative Dateien (REL):
  In relativen Dateien liegen die Daten für uns Anwender nicht hintereinander, sondern relativ zueinander vor. Hierbei wird beim Anlegen einer REL-Datei eine Datensatzlänge vorgegeben, die für ein REL-File immer gleich bleibt. Wenn Sie nun Daten in die Datei schreiben, werden selbige zu einem Datensatz zusammengefasst und mit einer fortlaufenden Satznummer versehen. Wenn Sie zum Beispiel die Adressen Ihrer Adressverwaltung in relativen Datensätzen speichern, und sich merken, unter welchen Datensatznummern Sie die einzelnen Adressen wiederfinden, so können Sie direkt auf einen Datensatz zugreifen. Die Vorteile gegenüber sequentieller Speicherung liegen auf der Hand: durch den Direktzugriff sind die Daten erstens schneller erreichbar, weil nicht eine komplette Datei eingelesen werden muß, sondern eben immer nur ein einziger Datensatz, und sie können zweitens extern gelagert werden, so daß Sie kostbaren Arbeitsspeicher im 64er sparen. Die Programmierung von relativen Dateien wird uns im 2.Teil dieses Kurses noch näher beschäftigen.
- User-Dateien (USR)
  Diese Dateien sind speziell für die direkte Floppyprogrammierung gedacht. Da die 1541 ja über einen eigenen Prozessor verfügt kann dieser auch direkt in Maschinensprache programmiert werden. Legt man nun ein Assemblerprogramm in einem USR-File ab, so kann dieses von der Floppy direkt in ihren eigenen Arbeitsspeicher geladen, und dort von ihr ausgeführt werden.
- Deleted Files (DEL)
  Mit der Kennung "DEL" werden Files markiert, die von der Diskette gelöscht wurden. Die Floppy löscht nun ein File nicht wirklich, sondern sie versieht den File-Eintrag im Directory einfach mit der DEL-Kennung und gibt die vom File belegten Blocks schlichtweg als 'unbelegt' wieder frei. Das bedeutet, daß ein File nach seiner Löschung auch immer wieder restauriert werden kann, solange man keine neuen Daten auf die Diskette geschrieben hat. Ein mit DEL gekennzeichneter Eintrag ist normalerweise im Directory nicht sichtbar. Wie das dennoch möglich ist, und wie man ein gelöschtes File wieder retten kann, soll und auch in einer der nächsten Ausgaben dieses Kurses beschäftigen.

DIE PROGRAMMIERUNG SEQUENTIELLER DATEIEN

Speziell die SEQ und REL-Dateien sollen nun in diesem und dem nächsten Teil dieses Kurses zum Zuge kommen. Beginnen wir mit der Programmierung sequentieller Files.

Wie oben schon erwähnt, liegen in SEQ und PRG Dateien die Daten sequentiell, also Byte hinter Byte, vor. In der Reihenfolge, mit der wir Daten in ein solches File hineinschreiben, müssen wir sie auch wieder auslesen. Die Vorgehensweise hierbei ist denkbar einfach.
Zunächst wollen wir einmal ein File schreiben. Hierzu müssen wir lediglich ein SEQ–File zum Schreiben öffnen und dann mittels des PRINT#–Befehls Daten hineinschreiben. Hierzu ein Beispiel:

```
OPEN 1,8,2,"DATEN,S,W"

PRINT#1,ZA
PRINT#1,a$
PRINT#1,x%
PRINT#1,CHR$(0);CHR$(40);

CLOSE 1
```

Hier öffnen wir zunächst einmal ein File mit dem Namen "DATEN". Innerhalb der Namensangabe verlangt die Floppy nun noch eine Spezifizierung des File-Typs und der Datenoperation (Lesen oder Schreiben) die durchgeführt werden soll.
Das "S" steht dabei für eine Sequentielle Datei. Andere mögliche Filetypen wären "R" für relative,"P" für Programm- oder aber auch "U" für Userdateien. Die "P" Angabe nimmt dabei eine Sonderstellung ein. Da sie der am häufigsten benutzte Filetyp ist, muß sie nicht explizit angegeben werden. Wenn keine Typenangabe gemacht ist, nimmt die Floppy automatisch an, daß es sich um eine "PRG"-Datei handelt. Dies funktioniert jedoch nur, wenn eine der beiden Sekundäradressen 0 oder 1 benutzt wurde.
Dazu gleich mehr.
Direkt nach dem Filetyp kommt, ebenfalls durch ein Komma getrennt, eine Bezeichnung der Datenoperation, die durchgeführt werden soll. Im Beispiel benutzten wir die Operation "W", was für Write= schreiben steht. Insgesamt sind hier aber drei verschiedene Operanden möglich:

(R)ead	- zum Lesen einer Datei
(W)rite	- zum Schreiben einer Datei
(A)ppend	- zum Anhängen an eine alte Datei

Die "A"-Option entspricht im Prinzip der "W"-Option. Daten werden in einem File gespeichert, jedoch mit dem Unterschied, daß diese Daten an eine schon bestehende Datei angehängt werden, und nicht etwa wie bei " W" eine neue Datei auf der Diskette angelegt wird.
Nachdem im obigen Beispiel eine sequentielle Datei zum Scheiben geöffnet wurde, können wir Daten mittels des PRINT#-Befehls in sie hineinschreiben.
Dies kann in BASIC auf verschiedenste Arten geschehen, die Betriebssystemroutine des PRINT#-Befehls passt sich automatisch den Variablentypen an (wie Sie sehen wurden in obigem Beispiel String-, Float und Integervariablen geschrieben). Möchte man ein "echtes" Zeichen in ein File schreiben, so benutzt man in der Regel die CHR$-Funktion, die nur einzelne Bytes schreibt, in obigem Beispiel die Bytes mit den Werten 0 und 40 .
Wichtig ist die Angabe eines Semikolons nach der CHR$-Ausgabe. Dies bedeutet nämlich, wie beim normalen PRINT-Befehl auch, daß nach der "Ausgabe" auf Diskette die nächste Ausgabe direkt folgen soll. Andernfalls hängt BASIC nämlich automatisch ein "Carriage Return" (CR, ASCII-Wert 13) an die Ausgabe an. Bei der Ausgabe von "reinen" Bytes kann dies hinderlich sein, weshalb man bei CHR$- Ausgaben in der Regel ein ";" mit angibt.
BASIC-Variablen MÜSSEN ohne Semikolon ausgegeben werden, da der INPUT#-Befehl, mit dem man die geschriebenen Daten später wieder lesen muß das CR als Endmarkierung eines Wertes heranzieht. Einzelne Bytes liest man mit der GET#-Funktion, die ja sowieso immer nur ein einzelnes Byte einliest. Hier ein Beispiel, das die obig geschriebene Datei wieder einliest:

```
OPEN 1,8,2,"DATEN,S,R"
INPUT#1,ZA
INPUT#1,A$
```

```
INPUT#1,x%
GET#1,Z1$: GET#1,Z2$
CLOSE 1
```

Die zwei einzelnen Zeichen aus obigem Beispiel sind nun in den String-Variablen Z1$ und Z2$ gespeichert. Möchte man sie wieder in echte Byte-Werte umwandeln, muß man die ASC-Funktion wiefolgt benutzen:

```
Z1%=ASC(Z1$)
Z2%=ASC(Z2$)
```

Wie Sie sehen, haben wir diesmal auch beim OPEN-Befehl die "R"-Angabe gemacht, damit die Floppy weiß, daß wir Daten lesen wollen.
Was Sie bei der sequentilellen Fileprogrammierung unbedingt anmerken sollten ist, daß die Daten eben "sequentiell", also hintereinander, in einem File liegen. Sie müssen sie demnach haargenau in der Reihenfolge wieder einlesen, mit der Sie sie geschrieben haben. Dies ist gar nicht so selbstverständlich wie es aussehen mag, wie Sie im nächsten Monat bei der relativen Dateiverwaltung feststellen werden.
Haargenau so, wie wir ein SEQ-File gelesen und geschrieben haben, können Sie mit PRG-Files hantieren. Sie müssen lediglich das "S" im Filenamen beim OPEN-Befehl in ein "P" abändern. Ansonsten ändert sich nichts.
Pfiffig bei der Arbeit mit PRG-Files ist nun die Anwendung der "besonderen" Sekundäradressen 0 und 1. Wie ich oben schon beschrieben hatte, können wir damit der Floppy gleich schon mitteilen, daß wir ein PRG-File lesen oder schreiben wollen. Die explizite Angabe ",P,R" oder ",P,W" entfällt. Die Sekundäradresse 0 steht für "PRG-File lesen", die Adresse 1 für "PRG-File schreiben". Hier einfach einmal zwei Beispiele:

"OPEN 1,8,0,"MEINEDATEI"	(Öffnet das PRG-File "MEINEDATEI" zum lesen)
"OPEN 1,8,1,"MEINEDATEI"	(Öffnet das PRG-File "MEINEDATEI" zum schreiben)

Nun können Sie ganz normal mit INPUT# und GET# lesen, bzw. mit PRINT# schreiben. Ebenso müssen Sie Ihre Files natürlich wieder wie gewohnt mit CLOSE schließen.
Die Verkürzung über die Sekundäradressen wird häufig in Kopierprogrammen benutzt, weil man so nicht noch umständlich den Filenamen-Appendix anzuhängen hat (besonders in Assembler eine zwar einfache, aber lästige Arbeit).
Das war es dann für diesen Monat. Im nächsten Monat wollen wir und dann an die relative Dateiverwaltung wagen, die zwar komplizierter zu Programmieren ist, mit der jedoch auch sehr flexibel und schnell gearbeitet werden kann. Bis dahin ein allzeit "Gut Hack",

Uli Basters (ub).

Teil 2 – Magic Disk 06/92

Hallo zum zweiten Teil des Floppy-Kurses. In der vorletzten MD hatten einiges über den Floppybefehlskanal und die Programmierung sequentieller Files gelernt. In diesem Teil wollen wir uns nunmehr mit der relativen Dateiverwaltung beschäftigen, die zwar etwas komplizierter zu programmieren, dafür aber weitaus flexibler als die sequentielle Dateiverwaltung zu handhaben ist.

DIE RELATIVE DATENVERWALTUNG
Im ersten Teil des Floppy-Kurses hatten wir bei den von der Floppy unterstützten Filetypen auch die sogenannten REL-Files besprochen. Sie bezeichnen Dateien, die einen RELativen Aufbau haben. Was das bedeutet wollen wir nun klären.
Sicherlich erinnern Sie sich, daß die sequentiellen Files, wie der Name schon sagt Daten

"sequentiell", also Byte hinter Byte enthalten. Wenn wir Daten in einem solchen File gespeichert hatten, so mussten wir immer das komplette File in den Rechner einlesen, um die Daten weiterverwenden zu können. Bei relativen Files ist dieser Aufbau nun anders geregelt. Sie arbeiten mit sogenannten "Datensätzen", oder engl."Records". Im Prinzip kann man eine relative Datei mit einem externen Variablenfeld vergleichen. Wir geben der Floppy lediglich an, daß wir z.B. das 24. Element (sprich: Record) ansprechen wollen, und schon können wir es lesen oder schreiben. Bei einem sequentiellen File hätten wir die 23 Eintragungen vorher erst überlesen müssen, bis wir auf das gewollte Element hätten zugreifen können.
Die Vorteile von REL-Files liegen damit auf der Hand:

1. Schnellerer Zugriff, da wir nur die benötigten Daten ansprechen müssen.
2. Speicherersparnis im Rechner selbst, da alle Daten extern gelagert sind und kein Speicherplatz mit momentan nicht benötigten Daten belegt wird.

Jedoch hat die relative Dateiverwaltung auch einen Nachteil: da sie von BASIC aus nicht unterstützt wird, ist ihre Programmierung relativ umständlich. Dennoch lässt sich auch das lernen, und wer es einmal kann der wird merken, daß die Vorteile überwiegen.

DIE PROGRAMMIERUNG

Bevor wir überhaupt irgendetwas aus oder in ein relatives File lesen oder schreiben können müssen wir es erst einmal generieren. Überhaupt wird ein relatives File ganz und gar anders behandelt als ein sequentielles. So wird beim Öffnen nicht mehr zwischen "Lesen" und "Schreiben" unterschieden. Wir öffnen ein solches File ganz einfach um der Floppy anzuzeigen, daß wir es nun benutzen wollen. Ob wir nun lesen oder schreiben ist ganz egal. Die Floppy erkennt automatisch, wenn wir etwas schreiben oder lesen. Das heißt also, daß wir bei der Benutzung eines relativen Files immer auch den Floppybefehlskanal öffnen müssen. Über spezielle Befehle wird die Floppy dann über die folgende Operation informiert. Sie stellt dann, je nach Befehl, beim Lesen auf dem relativen Filekanal die gewünschten Daten bereit, bzw. erwartet auf diesem die Daten, die geschrieben werden sollen.
Wollen wir nun also einmal ein relatives File anlegen, damit wir es benutzen können. Bevor wir das tun, sollten wir uns überlegen, wie lang ein Datensatz unseres REL-Files werden soll, und wie viele davon wir vorläufig darin speichern wollen. Ersteres müssen wir deshalb tun, weil die Datensätze eines relatives Files immer gleich lang sein müssen, um der Floppy das Auffinden eines Datensatzes zu ermöglichen. Nehmen wir einfach einmal an, daß wir 300 Datensätze zu je 80 Zeichen anlegen wollen. Wie oben schon erwähnt, öffnen wir zuerst einmal den Floppybefehlskanal. Anschließend folgt der OPEN-Befehl für das relative File. Wir legen dabei wie gewohnt die logische Filenummer, die Gerätenummer und die Sekundäradresse fest, und geben einen Namen für unsere Datei an. An diesen angehängt folgt die Typenkennung ",L," sowie der Länge des Datensatzes in diesem File. Hier ein kleiner Hinweis: im ersten Teil dieses Kurses erwähnte ich, daß die Kennung für eine relative Datei beim Öffnen ein "R" ist.
Das war leider eine Fehlinformation! "L" ist die richtige Bezeichnung für den OPEN-Befehl!
Hier nun ein Beispiel, in dem wir das relative File "TEST" mit 80 Zeichen pro Datensatz eröffnen:

```
OPEN 1,8,15
OPEN 2,8,3,"TEST,R,"+CHR$(80)
```

Der Befehlskanal hat nun die logische Filenummer "1", das relative File die "2" . Wichtig beim Öffnen des letzteren ist auch die Wahl der Sekundäradresse, da diese bei der Befehlsübergabe an den Befehlskanal verwendet wird. Wählen Sie bei der Sekundäradresse bitte nicht die vorreservierten Nummern 0,1 und 15(siehe Floppy-Kurs, Teil 1), sondern nur Nummern zwischen 2 und 14. Nachdem wir nun also alles geöffnet hätten, was wir benötigen, müssen wir jetzt erst einmal die gewünschten 300 Datensätze in der REL-Datei anlegen. Das funktioniert eigentlich ganz einfach: wir sprechen lediglich mit einem speziellen Befehl den 300. Datensatz an, und schreiben den Wert 255 hinein. Die Floppy generiert in diesem Fall nämlich

automatisch alle fehlenden Datensätze - in unserem Beispiel also alle 300. Das Generieren dieser Sätze kann jetzt einige Zeit in Anspruch nehmen, da die Floppy nun den Speicherplatz, den sie für alle Sätze braucht, automatisch belegt und mit den Bytewerten 255 füllt. Stören Sie sich bitte nicht daran, wenn während dieses Arbeitsgangs (wie auch bei allen anderen Operationen mit relativen Files) die Floppy-LED zu blinken beginnt, oder trotz Zugriffs zeitweise erlischt. Das ist ganz normal bei der Arbeit mit relativen Files.
Wenn die Floppy mit dem Anlegen der relativen Datei fertig ist blinkt sie übrigens sowieso, da wir durch den Zugriff auf einen noch nicht existierenden Datensatz einen "Record not present" Fehler erzeugt haben, der uns jedoch nicht weiter stören soll. Durch Auslesen des Befehlskanals stoppen wir das LED-Blinken. Hier nun das Ganze als Programm.

```
10  OPEN 1,8,15
20  OPEN 2,8,3,"TEST,R,"+CHR$(80)
30  HI=INT(300/256): LO=300-256*HI
40  PRINT#1,"P"+CHR$(3)+CHR$(LO)+CHR$(HI)+CHR$(1)
50  PRINT#2,CHR$(255)
60  INPUT#1,A,B$,C,D:
70  PRINT A,B$,C,D
80  CLOSE1: CLOSE2
```

Die OPEN-Befehle aus den Zeilen 10 und 20 kennen wir ja schon. In Zeile 30 spalten wir die Zahl 300 (die Anzahl der Datensätze, die wir in unserer Datei verwenden möchten) in Low- und High-Byte auf, da mit dem CHR$-Befehl ja immer nur 8-Bit-Werte (von 0 bis 255) übergeben werden können, und durchaus mehr Datensätze möglich sein können, müssen wir einen 16-Bit-Wert in Lo- / Hi-Folge an die Floppy senden. Dies geschieht in der folgenden Zeile - mit dem ersten PRINT#-Befehl senden wir den Positionierungsbefehl an die Floppy. Wohlgemerkt geschieht dies über den Befehlskanal!
Aus dem Beispiel ist die Syntax des Positionierungsbefehls ersichtlich. Beginnend mit dem Zeichen "P" (für "positionieren") werden in Reihenfolge die Sekundäradresse der REL-Datei auf die sich die Positionierung beziehen soll, die Datensatznummer in Lo/ Hi-Folge, sowie die Byteposition innerhalb des entsprechenden Datensatzes mittels CHR$-Codes an die Floppy übermittelt. In Zeile 50 wird nun über den logischen Kanal des REL-Files der Wert 255 in den positionierten Datensatz geschrieben. Da er, wie alle anderen vor ihm, noch nicht existiert, beginnt die Floppy nun damit alle Datensätze anzulegen, um den Schreibbefehl in Record 300 ausführen zu können. Es ist übrigens wichtig, daß Sie beim Erzeugen einer REL-Datei den Wert 255 schreiben, weil dieser nämlich als Endmarkierung beim Lesen dient. Hierzu jedoch später mehr.
In den Zeile 60 und 70 lesen wir nun noch den Fehlerkanal aus und geben die " Record not present"-Fehlermeldung aus, um die blinkende Floppy-LED zu löschen und schließen anschließend die beiden offenen Files - schon haben wir eine REL-Datei zur Verfügung!

DER SCHREIBZUGRIFF
Möchten wir nun mit unserer selbst erstellten REL-Datei arbeiten, so müssen wir sie natürlich öffnen. Hierbei ist darauf zu achten, daß wir dieselbe Datensatzlänge angeben, wie wir sie beim Erzeugen der Datei verwendet haben. Andernfalls kommt die Floppy nämlich mit der Verwaltung der Datensätze durcheinander, was verheerende Folgen bei Schreibzugriffen haben kann. Benutzen Sie am Besten also den gleichen OPEN-Befehl, den Sie auch beim Erstellen benutzt haben!!!
Wenn wir jetzt etwas in unsere Datei schreiben möchten, so verfahren wir im Prinzip genauso, wie beim Erstellen der Datei (denn das war ja nichts anderes als das Schreiben in eine REL-Datei).
Wir öffnen also zunächst Befehlskanal und REL-Datei, positionieren mittels Befehlskanal auf den gewünschten Datensatz und schreiben über die logische Filenummer der REL-Datei Daten in diesen Satz hinein. Hier ein Beispiel:

```
10  OPEN 1,8,15
20  OPEN 2,8,3,"TEST,R,"+CHR$(80)
30  PRINT#1,"P"+CHR$(3)+CHR$(1)+CHR$(0)+CHR$(1)
40  PRINT#2,"DIESER TEXT WIRD NUN IN DATENSATZ NUMMER EINS GESPEICHERT!";
50  CLOSE1: CLOSE2
```

Im Positionierbefehl wird wieder die Sekundäradresse 3 verwendet. Diesmal positionieren wir jedoch auf Byte 1 des ersten Datensatzes unserer Datei "TEST" (Die Werte 1 und 0 entsprechen der LO/ HI-Darstellung der Zahl 1-->256*0+1=1). In Zeile 40 wird dann mit einem ganz normalen PRINT#-Befehl ein Text in den positionierten Datensatz geschrieben. Da der Datensatz diesmal schon existiert brauchen wir demnach auch keinen Fehler von der Floppy auszulesen, da voraussichtlich keiner auftritt.
Anstelle des Textes könnte natürlich auch eine Stringvariable stehen. Achten Sie bitte darauf, daß Sie nie längere Texte in einen Datensatz schreiben, als selbiger lang ist, da Sie sonst einen Teil der Daten im folgenden Datensatz verlieren könnten.
Wichtig ist auch, ob Sie beim Schreiben das Semikolon (";") verwenden, oder nicht. Verwenden Sie es nicht, so können Sie beim Lesen den INPUT#-Befehl verwenden. Dieser erkennt das Ende eines Lesevorgangs immer an einem "Carriage Return Code"(kurz "CR"= CHR$(13)), der von einem PRINT# OHNE Semikolon immer automatisch gesendet wird. In dem Fall müssen Sie aber auch bedenken, daß ein Text den Sie schreiben nie länger als die Datensatzlänge-1 sein darf, da das CR ebenfalls als ganzes Zeichen in den Datensatz geschrieben wird.
Nun ist, wie wir später sehen werden, das Lesen mittels INPUT# nicht immer von Vorteil, weshalb Sie einen Text auch MIT Semikolon schreiben können. In dem Fall müssen wir später beim Lesen eine GET#–Schleife verwenden, da keine Endmarkierung (das "CR") für den INPUT#–Befehl geschrieben wurde.

DER LESEZUGRIFF
Auch der Lesezugriff weicht nicht sonderlich von den bisherigen Beispielen ab. Wie immer öffnen die beiden Floppykanäle und positionieren auf den gewünschten Datensatz. Nun haben wir zwei Möglichkeiten unsere Daten wieder auszulesen:
Wurden die Daten OHNE Semikolon geschrieben, so genügt ein einfaches "INPUT#2,A$" um unseren Text wieder zu Lesen und in A$ abzulegen.
Wurden Sie MIT Semikolon geschrieben, so müssen wir den umständlicheren Weg über eine GET#-Abfrage gehen. Dazu zwei Anmerkungen:

1. Bei der GET#–Abfrage sollten nicht mehr Zeichen gelesen werden, als maximal in dem Datensatz vorhanden sein können. Bei einer Länge von 80 Zeichen wird die GET#-Schleife also nicht mehr als 80 mal durchlaufen.
2. Was tun wir, wenn der Datensatzinhalt kürzer ist, als die festgelegte Datensatzlänge? Wie ich oben schon einmal erwähnte, dient der Byte-Wert 255 als Endmarkierung innerhalb einer REL-Datei. Dies stellt sich so dar, daß ein leerer Datensatz alle Bytes mit dem Wert 255 gefüllt hat. Schreiben wir nun einen Text in diesen Datensatz, so werden alle benötigten Zeichen mit dem Text überschieben.

Das darauf folgende Zeichen enthält dann aber immer noch den Wert 255.
Dementsprechend können wir sagen, daß wenn wir beim Lesen eines Strings dieses Zeichen erhalten, der String zu Ende sein muß.
Durch diese beiden Punkte ergibt sich also folgende Schleife zum Lesen eines Strings:

```
 90  ...
100  A$=""
110  FOR i=1 TO 80
120  GET#2,B$
130  IF ASC(B$)=255 THEN 160
140  A$=A$+B$
```

```
150 NEXT
160 ...
```

DATABANKING MIT RELATIVEN FILES

Sie sehen, daß das Arbeiten mit REL-Files trotz aller Gegenteiligen Voraussagen eigentlich relativ einfach ist.
Wir müssen jeweils nur richtig positionieren und können dann beliebig Lesen und Schreiben. Nun gibt es jedoch noch einige Kniffe, die man kennen sollte, wenn man effektiv mit relativen Dateien arbeiten möchte. Diese will ich nun ansprechen.

DATENFELDER:

Bei jeder Datenverarbeitung werden Sie meist mehrere Angaben in einem Datensatz machen. Einfachstes Beispiel ist hier eine Adressverwaltung. Hier müssen Sie pro Datensatz einen Namen, Strasse, Wohnort, Telefonnummer, etc. angeben, die Sie nachher auch immer wieder auf Anhieb in Ihrer Adressdatei finden müssen. Diese einzelnen Einträge in einem Datensatz nennt man Datenfelder. Wie bei relativen Files so üblich, sollte man sich dann jeweils auf eine maximale Länge eines Datenfeldes beschränken. Sie könnten nun für jedes Datenfeld eine eigene REL-Datei anlegen, also beispielsweise eine Datei mit 30 Zeichen pro Satz für alle Namen, eine mit 40 Zeichen für alle Straßen, eine mit 15 Zeichen für alle Telefonnummern, eine mit 4 Zeichen für alle Postleitzahlen usw. Diese Lösung birgt jedoch ein größeres Problem in sich: der C64 verwaltet nämlich immer nur EINE offene REL-Datei.
Das bedeutet in der Praxis, daß Sie jedesmal, wenn Sie eine komplette Adresse ausgeben wollen, alle Ihre REL-Files nacheinander öffnen, lesen und schließen müssen. Was das an Programmier- und Zeitaufwand beim Zugriff bedeutet ist verheerend. Deshalb geht man in der Regel einen anderen Weg. Halten wir doch einfach einmal an dem Beispiel der Adressverwaltung fest. Zunächst wollen wir uns einmal überlegen, welche und wieviele Felder wir verwenden wollen, und wie lang sie im einzelnen sein sollen. Für unsere kleine Adressverwaltung wollen wir 6 Felder pro Datensatz definieren:

```
1) "Name"        (20 Zeichen)
2) "Vorname"     (15 Zeichen)
3) "Strasse"     (30 Zeichen)
4) "Postleitzahl"  ( 4 Zeichen)
5) "Ort"         (30 Zeichen)
6) "Telefon"     (15 Zeichen)
```

Kommen wir nun zu dem Lösungsweg, denn man hier in der Regel geht. Anstelle von 6 einzelnen Dateien legen wir nun eine einzige Datei an, in der in einem Datensatz jeweils alle 6 Felder abgelegt werden. Dadurch ergibt sich eine Datensatzlänge von 114 Zeichen (20+15+30+4+30+15=114). Wir können nun wiederum zwei Wege gehen, mit denen wir die sechs Felder in einem Datensatz speichern. Ich gehe dabei davon aus, daß die Einträge vom Programm schon abgefragt wurden und in Stringvariablen stehen:

Die einfachere, dafür jedoch unflexiblere, Methode sieht folgendermaßen aus: Wir schreiben mit mehreren PRINT#-Befehlen OHNE Semikolon alle Stringvariablen hintereinander in ein Datenfeld hinein. Später können wir sie genauso wieder mittels INPUT# einlesen. Hierbei ist es egal, wie lang ein Feldeintrag ist, solange er die vorgegebene Länge nicht überschreitet (wäre das bei allen Feldern nämlich der Fall, so würden wir mehr Zeichen in einen Datensatz schreiben, wie dieser lang ist, was man tunlichst unterlassen sollte). Je nach dem, wie flexibel unser Adressverwaltungsprogramm sein soll entstehen nun jedoch diverse Schwierigkeiten. So müssen wir zum Beispiel immer alle Felder eines Datensatzes einlesen, wenn wir eine Datei z. B. nach einem einzelnen Feld sortieren möchten. Für gerade diese Aufgabe werden dann immer 5 Felder zuviel gelesen, was sich wiederum auf die Verarbeitungszeit RELativ auswirkt. Das zweite Problem ist die Endmarkierung nach jedem Feldeintrag. Wie oben ja schon

dargestellt müssen wir bei dieser Methode OHNE Semikolon arbeiten, und in dem Fall hängt PRINT# immer ein 'CR' an einen String an. Dadurch müssen wir von den obigen Feldlängen jeweils ein Zeichen abziehen (der Name z. B. darf nicht mehr 20, sondern nur noch 19 Zeichen lang sein). Sie sehen also, daß diese Methode zwar einfacher zu programmieren, aber sicherlich unflexibler ist.

Kommen wir zu der flexibleren Methode.
Hierbei halten wir wie oben an den Feldlängen fest. Da wir nun ja wissen, wie lang ein Feldeintrag maximal sein kann, können wir im Prinzip vorausberechnen, an welcher Stelle im Datensatz welches Feld zu finden, bzw. abzulegen ist, ohne daß sich zwei Felder überschneiden. Hierbei müssen wir darauf achten, daß entweder ein Feld immer so lang ist, wie es maximal sein darf (bei kürzeren Einträgen wird der Rest einfach mit SPACE-Zeichen aufgefüllt), oder aber wir verwenden den Trick mit der GET#-Abfrage beim Lesen. Letzteres ist wohl die eleganteste Lösung, da sie weniger Programmieraufwand erfordert und gleichzeitig die Lese- und Schreibzugriffe beschleunigt, da immer nur so viele Zeichen gelesen werden, wie auch wirklich benötigt werden (und nicht immer die maximale Feldlänge). Wollen wir nun einmal ausrechnen, an welchen Bytepositionen die einzelnen Felder abgelegt werden:

Feldname	**Beginn**	**Länge**
Name	1. Byte	20 Byte
Vorname	21. Byte	15 Byte
Straße	36. Byte	30 Byte
PLZ	66. Byte	4 Byte
Ort	70. Byte	30 Byte
Telefon	100. Byte	15 Byte

Anhand der Länge eines Feldes können wir immer die erste Position des folgenden Feldes berechnen, indem wir die Anfangsposition im Datensatz mit der Feldlänge addieren.
Nun wissen Sie ja, daß man beim Positionierbefehl nicht nur die Datensatznummer, sondern auch die Byteposition innerhalb des gewählten Datensatzes angeben kann. Und über diese Methode können wir nun ganz bequem jede der 6 Feldpositionen einstellen und den Feldeintrag hineinschreiben. Beim Lesen können wir, da wir für jedes Feld ja die Anfangsposition innerhalb eines Datensatzes kennen, dieses ebenfalls direkt anwählen und Lesen. Das ist vor allem beim Sortieren einer REL-Datei von Vorteil, da wir nun nach allen sechs Feldern eine Datei beliebig sortieren können, ohne die übrigen fünf Felder noch extra einlesen zu müssen. Das erhöht die Arbeitsgeschwindigkeit ungemein. Ein anderer Geschwindigkeitsvorteil ergibt sich beim Ändern von Datensätzen. Wollen wir zum Beispiel in einer Adressdatei nur die Straße verändern, weil die betreffende Person umgezogen ist, so genügt es auf das Feld "Straße" (Byte 36 im entsprechenden Datensatz) zu positionieren und den neuen Eintrag hineinzuschreiben. Hierbei müssen Sie jedoch auch beachten, daß der neue Straßenname unter Umständen kürzer ist, als der alte. Wir müssen dann nämlich noch ein CHR$(255) nachschicken, damit unsere GET#-Schleife auch ihre Endmarkierung findet. Andernfalls würde Sie die restlichen Zeichen des alten Straßennamens mitlesen.

Ein Beispiel:

Alte Straße	→	"Bahnhofstrasse 76"
Neue Straße	→	"Am Brunnen 2"
Ergebnis beim Lesen OHNE eine neue Endmarkierung	→	"Am Brunnen 2se 76"

Schreiben Sie aber auch bitte nur dann ein CHR$(255) danach, wenn der neue Straßenname

kleiner als die maximale Feldlänge ist. Andernfalls würden Sie wieder über die Feldgrenze hinaus schreiben und so das erste Zeichen des darauffolgenden Feldes überschreiben!

SORTIEREN ODER INDIZIEREN:
Oben habe ich schon einmal die Möglichkeit der Sortierung einer Datenbank angesprochen. Die augenscheinlich einfachste Art der Sortierung wäre wohl das alphabetische Ordnen der Einträge eines Feldes (z. B. des Namensfeldes), und das anschließende Umkopieren der Datensätze, so daß im ersten Datensatz auch wirklich der alphabetisch erste und im letzten Datensatz der alphabetisch letzte Name steht. Diese Lösung verwendet aber wohl keiner, da man sich vorstellen kann, daß die Sortierung durch das Umkopieren der Datensätze einen extremen Zeitaufwand bedeutet (gerade bei einfachen Sortieralgorithmen ist die Anzahl der Austausche zwischen zwei Einträgen ungemein hoch) . Zusätzlich erfordert diese Methode nach jedem Neueintrag das Umkopieren der kompletten REL-Datei, da ein neuer Datensatz ja ebenfalls irgendwo einsortiert werden muß, und damit alle folgenden Datensätze einen Datensatz weiter nach hinten rücken. Lassen Sie uns also diese Methode in dem Mülleimer werfen und die schnelle, komfortable und flexible Methode herauskramen:" Indizierung" heißt das Zauberwort!
Im Prinzip bedeutet dieses Wort nicht mehr als "Sortieren", jedoch ist die Handhabung etwas anders. Hier möchte ich noch einmal auf das Beispiel der Adressverwaltung zurückgreifen. Gehen wir also davon aus, daß wir unsere Adressen nach den Namen der eingetragenen Personen ordnen wollen. Zu diesem Zweck lassen wir unseren Sortieralgorithmus nach und nach alle Namen aus der REL-Datei auslesen und alphabetisch in eine Liste einordnen. Dabei wollen wir uns zu jedem Namen auch seine Datensatznummer merken.
Letztere können wir dann als Referenz auf den alphabetisch richtigen Datensatz verwenden. Ich möchte Ihnen dies anhand eines Beispiels verdeutlichen. Nehmen wir einmal eine Adressdatei, die die folgenden vier Namenseinträge, in der Reihenfolge in der sie eingegeben wurden, enthält:

Satznummer	Eintrag (Name)
1	Müller
2	Becker
3	Schmidt
4	Meier

Nun sortieren wir unsere Datei nach Namen und erhalten folgende Reihenfolge:

Satznummer	Eintrag (Name)
2	Becker
4	Meier
1	Müller
3	Schmidt

Die alphabetisch richtige Reihenfolge legen wir nun z.B. in einem Variablenfeld ab. Dieses Variablenfeld nennt man Index. Wenn wir nun den alphabetisch ersten Eintrag lesen wollen, so schauen wir uns einfach den ersten Eintrag in der Indexliste an, und lesen den Datensatz ein, der dort steht - im Beispiel also den zweiten.
Damit wir nicht jedes mal neu sortieren müssen ist es ratsam die Indexliste in einem sequentiellen File auf Diskette abzuspeichern. Wenn wir nun jedesmal, wenn die Adressverwaltung gestartet wird das Indexfeld einlesen, so können wir ganz einfach und schnell

alphabetisch geordnete Datensätze finden und bearbeiten. Ebenso sollten wir darauf achten, daß neue Datensätze in die Indexliste richtig einsortiert werden und der Index wieder neu auf Diskette gespeichert wird.
Die Indizierung bietet jedoch noch weitere Vorteile. So können wir auch ganz beliebig nach verschiedenen Feldern sortieren und gleichzeitig beide sortierten Dateien weiterverwenden. Wenn Sie Ihre Adressen manchmal besser über den Vornamen finden können, so ist es sinnvoll auch nach dem Vornamen zu indizieren und diesen Index als Alternative Verwenden zu können. Beim Umschalten zwischen zwei Indizes müssen Sie dann jeweils die neue Indexdatei einlesen. Dies ist sogar bequem möglich, da wir zu der einen offenen REL-Datei auch noch eine SEQ-Datei öffnen dürfen, ohne daß der 64 er verrückt spielt. Wir können so als derzeit den Index wechseln, ohne dabei noch umständlicherweise die REL-Datei schließen und anschließend wieder öffnen zu müssen.

EIN WORT ZU NUMERISCHEN EINTRÄGEN:
Vielleicht wollen Sie irgendwann einmal auch numerische Einträge in einer REL-Datei speichern. Das wäre bei unserer Adressverwaltung zum Beispiel bei dem Postleitzahlenfeld sehr gut denkbar. In manchen Fällen kann uns das auch einen Speicherplatzvorteil im Gegensatz zu der Speicherung als String geben. Hierbei müssen wir jedoch immer gewisse Regeln beachten, sonst werden Sie ganz schön Probleme mit den Datensatzlängen bekommen. Schreiben Sie nämlich eine numerische Variable (wie z.B."A" als Float-, oder "A%" als Integervariable) mittels "PRINT#2, A"(oder "PRINT#2,A%") in eine Datei, so wandelt die PRINT#-Routine Ihre Variable immer in eine Zeichenkette, also eine String, um. Wenn diese numerische Variable nun verschieden viele Stellen besitzen kann, so werden Sie ganz schöne Schwierigkeiten bekommen.
Logischerweise hat die Zahl "100" eine Stelle weniger als die Zahl "1000". Hinzu kommt, daß auch noch ein "-" als Vorzeichen davor stehen kann und daß der PRINT#-Befehl immer noch ein SPACE-Zeichen (" ") vor einer Zahl ausgibt, oder daß ganz kleine oder ganz große Zahlen in der wissenschaftlichen Schreibweise (also z. B."3.456+ E10") ausgegeben werden.
Wie lang sollen wir dann unser Feld nun machen? Eine Antwort darauf ist schwierig, da das dann immer auch ganz von dem Verwendungszweck abhängt. Entweder müssen Sie dem Benutzer schon bei der Eingabe der Zahlen gewisse Einschränkungen auferlegen, oder aber durch eigene Umkonvertierung einer Zahl eine bestimmte Bytelänge festlegen können.
Letzteres möchte ich hier als Beispiel aufzeigen. Wir wollen uns auf positive Integerzahlen beschränken, also ganze Zahlen (ohne Nachkommastellen), die von 0 bis 65535 gehen. In diesem Fall genügt es eine solche Zahl in Low- und Highbyte aufzuspalten und die beiden CHR$-Codes zu speichern. In der Praxis sieht das ähnlich wie in der Syntax des Positionierungsbefehl für REL-Dateien aus:

```
ZA=1992
HI=INT(ZA/256): LO=ZA-256*LO
PRINT#2,CHR$(LO);CHR$(HI);
```

Damit können Sie nun alle oben genannten Zahlen schreiben. Aber eben nur diese.
Möchten Sie z. B. Float-Zahlen schreiben, so sollten Sie sich überlegen, wie groß oder klein diese maximal sein können.
Dann können Sie sie sich einmal mit dem normalen PRINT-Befehl auf dem Bildschirm ausgeben lassen und die Stellen, die sie einnehmen abzählen. Wenn Sie sicher sein können, daß dies die maximale Länge einer Ihrer Zahlen ist, so können Sie sie als Feldlänge definieren. Beachten Sie aber auch immer, daß der PRINT#-Befehl vor einer Zahl immer ein Leerzeichen druckt und daß zusätzlich immer noch ein Minus-Zeichen vor einer Zahl stehen kann. Außerdem können Sie solche Zahlen auch nur mittels des INPUT#-Befehls wieder einlesen, weshalb Sie noch ein Zeichen mehr für das CR mitrechnen sollten.
Ein anderer Weg um Float-Zahlen zu speichern wäre das direkte übernehmen der 5-Byte-

Mantissen-Darstellung, wie das Betriebssystem des 64ers sie benutzt, jedoch ist diese Möglichkeit relativ kompliziert, da sie den intensiven Gebrauch von Betriebssystemroutinen erfordert, wovon wir momentan die Finger lassen wollen (in einer der späteren Folge: dieses Kurses werde ich aber noch einmal auf dieses Thema zurückkommen).

So. Das wäre es dann wieder einmal für diesen Monat. In der nächsten Ausgabe der "Magic Disk" wollen wir uns mit den Direktzugriffsbefehlen der Floppy befassen, die es uns ermöglichen, auf die Diskettenblocks direkt zuzugreifen.

(ub)

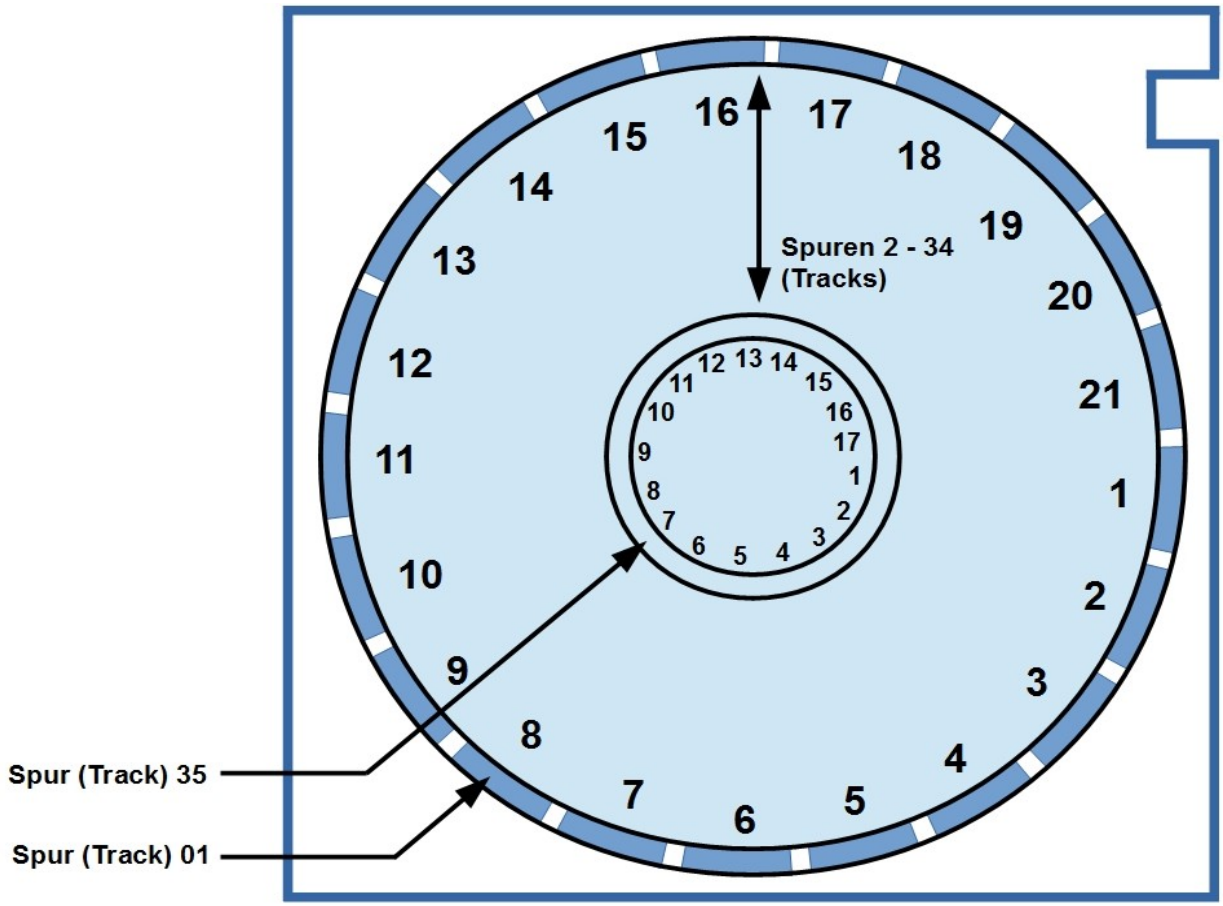
Teil 3 – Magic Disk 07 & 08/92

Hallo und herzlich Willkommen zum dritten Teil dieses Kurses. Nachdem wir uns in den ersten beiden Teilen um die sequentielle und die relative Dateiverwaltung gekümmert hatten, wollen wir nun in die tieferen Ebenen der Floppy-Programmierung einsteigen. Diesen Monat soll es um die Direktzugriffsbefehle gehen, mit denen wir Daten auf einer Diskette direkt manipulieren können.
Dabei haben wir die Möglichkeit auf die einzelnen Datenblöcke selbiger zuzugreifen und sie unseren Wünschen entsprechend zu verändern. Bevor wir beginnen, sollten erst einmal ein paar Grundlagen geklärt werden.

DIE DISKETTENSTRUKTUR
Zunächst wollen wir uns einmal die Grundstruktur einer Diskette anschauen. Nachdem sie mit dem Format-Befehl der Floppy formatiert wurde, enthält Sie 35 konzentrisch angeordnete Spuren (engl. "Tracks") und pro Spur 17-21 Sektoren. Diese Anordnung ist typisch für das 1541- Format. Die unterschiedliche Anzahl von Sektoren pro Spur ergibt sich aus der physikalischen Anordnung der Spuren. Dadurch, daß die Spuren von außen nach innen immer kleiner werdenden Kreisen entsprechen, wird demnach auch der Kreisumfang, oder die "Länge" einer Spur immer kleiner. Je kleiner nun die Spurlänge ist, desto weniger Sektoren haben auf ihr Platz. Die folgende Grafik über den Aufbau einer Diskette soll Ihnen das verdeutlichen...
Daraus ergibt sich nun die folgende Sektorenverteilung:

Spuren	Sektoren pro Spur
1 – 17	21
18 – 24	19
25 – 30	18
31 – 35	17

Wichtig ist, daß Sie wissen, wieviele Sektoren ein bestimmter Track hat, da wir diese Information bei den Direktzugriffsbefehlen benötigen, um einen speziellen Block anzusprechen. Versuchen Sie nun aber Sektor 19 von Spur 35 zu lesen, so erhalten Sie natürlich eine Fehlermeldung von der Floppy zurück (obwohl es einen Sektor 19 bei den Tracks 1-24 gibt!).



DIE FLOPPY - EIN EIGENSTÄNDIGER COMPUTER

Wie ich vielleicht schon einmal erwähnte stellt die 1541 einen vollblütigen, unabhängigen Computer dar. Sie verfügt über einen eigenen 8- Bit-Prozessor (den 6502, der Vorgänger des 6510 aus dem 64 er, der auch im VC20 Verwendung fand), zwei I/ O Bausteine, die man auch VIA nennt (Typ 6522, das Vorläufermodell des 6526-CIA-Chips, wie er auch zweimal im C64 vorhanden ist, und ebenfalls im VC20 eingesetzt war), einem 16-KB-ROM mit dem Betriebssystem (DOS) und 2 KB RAM als Arbeitsspeicher für letzteres und Zwischenspeicher für Diskettenblocks. Die VIA-Bausteine übernehmen dabei den Datentransfer mit dem 64er, sowie die Bedienung der Floppymechanik. Natürlich können Sie dem Floppyprozessor auch ein Programm geben, das er abarbeiten soll.

Dieses muß in Maschinensprache geschrieben sein, wobei der Befehlssatz des 6502 identisch mit dem des 6510 (aus dem 64 er) ist. Die Speicheraufteilung der Floppy sieht folgendermaßen aus:

Adresse	Belegung
$0000-$0800	2 KB RAM
ab $1800	VIA1(serieller Bus)
ab $1C00	VIA2(Laufwerkssteuerung)
$ C000-$ FFFF	Betriebssystem-ROM (16 KB)

Alle übrigen Bereiche sind unbelegt. Wie beim C64 auch, können Sie mit einem Programm, daß sich im Floppy-RAM befindet die Betriebssystemroutinen der Floppy aufrufen (so funktionieren z.B. einige Software-Floppyspeeder).

Kommen wir nun zum RAM der Floppy, denn das soll uns eigentlich interessieren. Ein Datenblock auf einer Diskette ist, wie Sie wissen 256 Bytes lang. Diese Datenlänge wird auch als Einheit für das Floppy-RAM benutzt: 256 Byte entsprechen einer sogenannten "Page"

(englisch "Seite" – den Assemblerprogrammierern unter Ihnen sicherlich ein geläufiger Begriff). Das Floppy-RAM wird nun in acht Pages unterteilt, die von 0 bis 7 durchnummeriert sind. Die Pages 0,1 und 2 sollten wir dabei außer Acht lassen, da sie für die Zeropage, den Stack und den Befehlspuffer der Floppy vorreserviert sind. Die Pages 3 – 7 sind Datenpuffer für Diskettenblöcke. Wir werden sie von nun an "Puffer" nennen.

DIE DIREKTZUGRIFFSBEFEHLE

Kommen wir nun endlich zu den Blockbefehlen selbst. Eigentlich gibt es drei Gruppen, von Befehlen, mit denen wir Abläufe innerhalb des "Floppy-Computers" steuern können: die Blockbefehle zum lesen, schreiben, belegen und freigeben von Blocks, die Memorybefehle, mit denen wir den Floppy-Speicher manipulieren können und die User-Befehle, die eine Verkürzung von Blockund Memorybefehlen darstellen.
Bevor wir nun einen dieser Befehle benutzen können, müssen wir natürlich den Floppybefehlskanal öffnen. Da wir aber immer auch einen Arbeitspuffer im Floppy-RAM benötigen, um die Direktzugriffsbefehle (einige Blockbefehle und zwei Userbefehle) anwenden zu können, müssen wir gleichzeitig auch einen Puffer für uns vorreservieren. Entweder überlassen wir dabei die Wahl des Puffers der Floppy, oder aber wir bestimmen die Puffernummer selbst. Letztes wird wohl nur in ganz besonderen Fällen vonnöten sein, weshalb wir uns auf die erste Methode festlegen wollen. Das Öffnen des Befehlskanals mit der Zuweisung eines Arbeitspuffers geschieht über das Öffnen eines eigenen Datenkanals, der für den Datentransfer mit dem Puffer herangezogen wird. Dies geschieht folgendermaßen:

```
OPEN 1,8,15       :REM Befehlskanal öffnen
OPEN 2,8,2,"#"    :REM Pufferkanal öffnen
```

Hier öffnen wir also wie gewohnt den Befehlskanal, dem wir die logische Filenummer 1 zuordnen. Der zweite OPEN-Befehl öffnet einen Kanal zur Floppy, den wir zum Auslesen und Schreiben unseres Datenpuffers verwenden werden. Das Doppelkreuz ("#") zeigt der Floppy an, daß wir mit einem Datenpuffer arbeiten möchten. Welche Nummer dieser hat, sei der Floppy überlassen. Sie sucht nun einfach einen freien Puffer aus, und ordnet ihm unserem Kanal zu. Möchten wir wissen, welcher Puffer ausgewählt wurde, so können wir seine Nummer direkt nach dem Öffnen durch Lesen des Pufferkanals erfahren:

```
GET#2,PN$
PRINT ASC(PN$)
```

Hierbei ist es wichtig, daß Sie den GET-Befehl innerhalb eines Programms benutzen, da er vom Direktmodus aus nicht verwendbar ist.
Wenn wir gleich einen bestimmten Puffer für uns reservieren möchten, so müssen wir beim OPEN-Befehl einfach die Puffernummer mit übergeben. Folgender Befehl reserviert den Puffer Nummer 3 (Floppy-Adressbereich $0300-$03FF) für uns:

```
OPEN 2,8,2,"#3"
```

Wichtig ist, daß wir uns die Sekundäradresse merken, mit der wir den Pufferkanal geöffnet hatten. Sie wird später dazu verwendet, um der Floppy mitzuteilen auf welchem Datenkanal Sie Daten für uns bereitzustellen hat, bzw. von uns empfangen wird (ähnlich den Befehlen für die RELative Dateiverwaltung). Wie üblich wählen wir bei der Sekundäradresse eine der freien Nummern von 2-14.

DIE BLOCKBEFEHLE

Nachdem wir nun einen Datenpuffer reserviert haben und einen Filekanal für diesen Puffer offen halten, können wir uns jetzt mit den Befehlen beschäftigen, die wir der Floppy geben können. Die erste Gruppe dieser Befehle sind die Blockbefehle. Mit ihnen können wir die einzelnen Blocks einer Diskette " hardwaremäßig" ansprechen. Aufgrund eines Blockbefehls

wird physisch auf den Block zugegriffen und in den Puffer eingelesen, bzw. aus ihm heraus geschrieben. Für die Beispiele innerhalb der Befehlsbeschreibungen gelten die obig definierten logischen Filenummern (Befehlskanal=1, Pufferkanal=2), sowie die Sekundärdresse 2 für den Pufferkanal.

1. BLOCK–READ ("B–R")
   Mit diesem Blockbefehl lesen wir einen Diskettenblock in unseren Datenpuffer ein. Hierbei müssen wir die Sekundäradresse unseres Pufferkanals, die Drivenummer (immer 0), den Track und den Sektor als Parameter übergeben. Hier die allgemeine Syntax:

   ```
   PRINT#BFN,"B–R";SA;DN;TR;SE
   ```

   Hierbei gilt:
   BFN = logische Filenummer des Befehlskanals
   SA = Sekundäradresse des Pufferkanals
   DN = Drivenummer (immer 0)
   TR = Tacknummer
   SE = Sektornummer

(Diese Variablennamen und Abkürzungen werde ich auch im Folgenden weiterverwenden.) Mit der folgenden Anweisung wird also Sektor 1 von Track 18 in unseren Puffer eingelesen. Von dort aus können Sie Ihn nun mittels einer GET-Schleife auslesen:

```
...
100  PRINT#1,"B-R 2 0 18 1"
110  FOR I=0 TO255
120  GET#2,a$: PRINT ASC(a$)
130  NEXT
…
```

2. BLOCK–WRITE ("B-W")
   Mit diesem Blockbefehl schreiben wir einen Block aus unserem Datenpuffer auf die Diskette. Seine Syntax ist analog der von "B–R", nur daß diesmal der Block geschrieben wird:

   ```
   PRINT#BFN,"B–W";SA;DN;TR;SE
   ```

3. BUFFER–POINTER ("B–P")
   Für jeden Datenpuffer der Floppy existiert ein Zeiger, der auf das aktuelle Byte im Puffer zeigt. Möchte man nun ein ganz spezielles Byte aus dem Puffer auslesen, so muß nicht unbedingt der ganze Puffer gelesen werden, um das gewünschte Byte zu erhalten. Sie können mit dem "B–P"–Befehl den Pointer auf selbiges positionieren und anschließend auslesen. Als Parameter benutzen Sie ganz einfach die Sekundäradresse des Pufferkanals und die Nummer des gewünschten Bytes minus 1. Möchten Sie z.B. das 200. Byte auslesen, so gehen Sie bitte wiefolgt vor:

   ```
   PRINT#1,"B-P 2 199"
   GET#2,A$: A=ASC(A$)
   ```

   Die zweite, oben angeführte Zeile liest nun das 200. Byte in eine Stringvariable und speichert es als Zahl in der Variablen "A".

4. BLOCK–ALLOCATE ("B-A")
   Mit diesem Floppybefehl können Sie gezielt einzelne Disketten-Blocks als "belegt" kennzeichnen. Das ist dann notwendig, wenn Sie Daten mittels "B–W" auf einer Diskette speichern, zusätzlich aber noch normale DOS-Files darauf ablegen wollen. Wird ein

Block nämlich nicht als belegt gekennzeichnet, so wird er beim Speichern eines DOS-Files unter Umständen überschrieben. Die belegten Blocks sind in der "Block–Allocation–Map" (BAM) einer Diskette gespeichert, die sich in Block 18,0 befindet. Mit dem Block–Allocate–Befehl wird hier dann der entsprechende Block als belegt eingetragen. Mehr zum Aufbau der BAM werden wir im nächsten Teil des Floppy-Kurses lernen. Die allgemeine Syntax des " B–A"–Befehls lautet:

PRINT#BFN,"B-A"; DN; TR; SE

Möchten Sie also z. B. den Block 5 von Spur 34 als belegt kennzeichnen, so müssen Sie folgenden Befehl benutzen:

PRINT#1,"B–A 0 34 5"

5. BLOCK–FREE ("B–F")
   Dieser Befehl ist das Gegenstück zu Block–Allocate. Mit ihm geben Sie einen belegten Block wieder zur Benutzung frei. Die Syntax ist dabei identisch zu "B–A":

   PRINT#BFN,"B–F"; DN; TR; SE

   Um den obig belegten Block wieder freizugeben senden Sie den folgenden Befehl an die Floppy:

   PRINT#1,"B–F 0 34 5"

6. BLOCK-EXECUTE ("B–E")
   Dieser Befehl ist identisch mit Block-Read. Er beinhaltet jedoch zusätzlich, daß der Diskettenblock, der in den Puffer geladen werden soll, ein ausführbares Floppy–Programm ist, das nach dem Laden direkt angesprungen wird.

DIE MEMORYBEFEHLE

Diese Floppybefehle beziehen sich auf die Floppy selbst und haben mit dem Direktzugriff nur indirekt zu tun. Trotzdem sind sie nützlich zum Auswerten von Daten, weshalb sie hier aufgeführt sind:

1. MEMORY-READ ("M-R")
   Dieser Befehl entspricht dem PEEK–Befehl von BASIC. Mit ihm können Sie gezielt einzelne, oder mehrere Speicheradressen der Floppy auslesen. Die allgemeine Syntax lautet dabei wiefolgt:

   PRINT# BFN,"M–R";CHR$(LO);CHR$(HI);CHR$(N)

   Hierbei stehen "LO" und "HI" für Low- und Highbyte der Adresse die gelesen werden soll und "N" für die Anzahl Bytes (0-255), die ab dort übertragen werden sollen. Das Low- und Highbyte einer Adresse ermitteln Sie mit folgenden Formeln:

   HI= INT(Adresse/256) LO= Adresse-256* HI

   Alle Parameter müssen als CHR$–Codes, also direkt, übertragen werden, und dürfen nicht als ASCII-Codes (wie in den bisherigen Befehlen) erscheinen. Die zu lesenden Bytes werden ebenfalls als absoluter CHR$–Code auf dem BEFEHLSKANAL (!) bereitgestellt (nicht etwa auf dem Pufferkanal, da der interne Floppyspeicher nichts mit Datenblocks zu tun hat).
   Als Beispiel für die Verwendung von "M–R" sei folgendes Programm aufgeführt.
   Es liest den ASCII-Code der ID der zuletzt initialisierten Diskette aus dem Floppyspeicher aus. Dieser wird vom Floppy-Betriebssystem automatisch in den Speicherstellen 18 und 19 gespeichert (LO=18;HI=0;N=2):

```
10  OPEN 1,8,15
20  PRINT#1,"M-R"; CHR$(18);  CHR$(0);CHR$(2)
30  GET#1,I1$: GET#1,I2$
40  CLOSE 1
50  PRINT "ID DER ZULETZT BENUTZTEN DISKETTE: ";I1$;I2$
```

2. MEMORY–WRITE
   Mit diesem Befehl schreiben Sie Daten in den Floppy-Speicher. Er entspricht dem POKE-Befehl von BASIC. Auch hier über- tragen Sie zunächst die Adresse in Low- und Highbytedarstellung, sowie die Anzahl der zu schreibenden Bytes. Letztere hieraufhin gesandt. Hier die allgemeine Syntax:

```
PRINT# BFN,"M–W";CHR$(LO);CHR$(HI);CHR$(N);CHR$(W1);CHR$(W2);...;CHR$(Wn-1) ; CHR$(Wn)
```

3. MEMORY-EXECUTE
   Dieser Befehl dient dem Ausführen eines Assembler-Programms innerhalb der Floppy. Damit hat auch er einen Verwandten in BASIC: den SYS-Befehl. Als Parameter übergeben Sie hier einfach nur die Startadresse des aufzurufenden Assembler-Programms:

   ```
   PRINT#BFN,"M-E";CHR$(LO);CHR$(HI)
   ```

DIE USERBEFEHLE

Die dritte Gruppe der Floppy-Direkt–Befehle sind die Userbefehle. Sie stellen eine Abkürzung von einigen Befehlen aus den anderen beiden Gruppen dar. Die Userbefehle beginnen alle mit einem "U", gefolgt von einer Zahl zwischen 1 und 9. Die ersten beiden Userbefehle sind wohl die wichtigsten:

1. "U1"
   Der U1–Befehl ersetzt den Block–Read–Befehl. In der Praxis wird er öfter verwendet, da er gegenüber "B-R" einen Vorteil bietet. Wird mit "B-R" ein Block eingelesen und anschließend der Inhalt des Puffers vom 64er aus ausgelesen, so wird das erste Byte nicht mitgesandt.
   Möchten wir dieses nun aber auch lesen, so müssen wir den "U1"- Befehl benutzen.
   Er sendet es zusätzlich mit – im Prinzip ist der U1–Befehl also besser als "B-R".
   Die Syntax ist identisch mit der von "B-R":

   ```
   PRINT# BFN,"U1";SA;DN;TR;SE
   ```

   Ansonsten verhält sich alles so, wie auch bei "B-R".

2. "U2"
   Der zweite Userbefehl ist der U2–Befehl (Ähnlichkeiten mit dem Namen einer bekannten Rock-Band sind rein zufällig). Er ersetzt den "B-W" Befehl. Auch hier ist der Vorteil des ersten Bytes gegeben. Sie benutzen den U2–Befehl, wie den "B-W"–Befehl:

   ```
   PRINT#LFB,"U2";SA;DN;TR;SE
   ```

3. "U3-U8"
   Diese sechs Userbefehle ersetzen ganz bestimmte "M-E"–Kommandos. Wird einer dieser Befehle gesandt, so springt die Floppy in eine Sprungtabelle ab Adresse $0500 und führt das dort stehende Programm aus. In der Regel besteht selbiges aus einer Reihe von "JMP $XXXX"–Anweisungen, mit denen Sie nun auf einfache Weise, vom 64 er aus, selbstdefinierte Programme in der Floppy aktivieren können. Die Userbefehle 3-8 benötigen keine Parameter. Bei Benutzung der Befehle werden folgende Adressen jeweils angesprungen:

Befehl	Springt an Adresse
U3	$0500
U4	$0503
U5	$0506
U6	$0509
U7	$050C
U8	$050F

Damit ersetzen diese Userbefehle also "M-E"Befehle wie z.B.:

```
PRINT#BFN,"M-E";CHR$(9);CHR$(5);
```

Wenn Sie den obigen Befehl abkürzen wollen, so genügt auch ein einfaches:

```
PRINT# BFN,"U6"4)
```

4. "U9"
   Der "U9"–Befehl wurde eingebaut, um die Kompatibilität der 1541 zum VC20, dem Vorgänger des C64, zu wahren. Mit ihm können Sie die Floppy in den Modus des entsprechenden Rechners schalten. Dies geschieht über folgende Kombinationen des "U9"–Befehls:

   ```
   PRINT#BFN,"U9+"; schaltet in C64-Modus
   PRINT#BFN,"U9–"; schaltet in VC20-Modus
   ```

5. "U:"
   Mit diesem Userbefehl lösen Sie einen RESET in der Floppy aus. Das Floppysystem wird dann wieder in den Einschaltzustand zurückversetzt. Der "U:"–Befehl entspricht einem "SYS 64738" beim C64.
   Auch er benötigt keine Parameter.

Das war es dann wieder für diese Ausgabe. In der nächsten Magic-Disk werden wir uns dann mit dem Aufbau von Files, der BAM und des Directorys beschäftigen. In diesem Zusammenhang werden wir dann die hier erlernten Floppy-Befehle in praktische Beispiele umsetzen.

(ub)

Teil 4 – Magic Disk 09/92

Hallo und herzlich willkommen zum vierten Teil unseres Floppy–Kurses. Nachdem wir das letzte Mal die Direktzugriffsbefehle kennengelernt haben, wollen wir uns dieses Mal mit der Struktur auf der Diskette beschäftigen. Dabei werden wir lernen, wie wir einzelne Diskettenblocks so manipulieren, daß zum Beispiel der Typ eines Files geändert ist, ein File nicht mehr gelöscht werden kann, etc.

WAS STEHT WO?

Sicherlich erinnern Sie sich noch daran, daß die 1541 eine Diskette in 35 Spuren zu je 17–21 Sektoren aufteilt. Wir wollen uns nun anschauen, wie das DOS (so nennt man das Betriebssystem der Floppy) nun Daten auf diesen Blocks ablegt. Dabei müssen wir gleich einmal unterscheiden zwischen DOSinternen Verwaltungsblöcken und den reinen Datenblökken. In ersteren stehen Informationen wie z. B. der Name der Diskette, ihre ID, welche Blocks belegt sind, und welche nicht, die Directoryeinträge, etc. Den

Verwaltungsblöcken ist ausschließlich der gesamte 18 . Track vorreserviert. Datenblöcke werden nie dort abgelegt, dazu dienen jedoch dann alle anderen Tracks (1–17 und 19–35).

DIE DATENBLÖCKE

Zunächst wollen wir uns mit der Struktur eines Datenblocks beschäftigen. Diese ist die einfachere, und demnach schneller erklärt. Dazu will ich Ihnen nun zunächst einmal beschreiben, was in der Floppy vorgeht, wenn ein File auf der Diskette gespeichert wird.
Zu aller erst erhält die Floppy vom C64 die Meldung, daß nun ein Schreibzugriff erfolgen soll. Gleichzeitig mit dieser Information wird der Filename des zu schreibenden Files, sowie sein Filetyp übertragen. Die Floppy schaut nun in den Blocks, die für die Directoryeinträge definiert sind nach, wo ein Eintrag noch frei ist, oder ob ein neuer hinzugefügt werden soll. Warum das so ist, wollen wir später bei den Verwaltungsblocks klären. Nehmen wir also einfach einmal an, daß Platz für einen Eintrag gefunden wurde, und daß der angegebene Filename noch nicht auf der Diskette existiert (dies wird nämlich ebenfalls festgestellt und bei Zutreffen wird mit der Fehlermeldung "File Exists" abgebrochen). Das DOS trägt nun im freien Eintrag den Namen und Typ des zu speichernden Files ein. Anschließend macht es sich auf die Suche nach einem freien Block in den Tracks 1–17, bzw.19–35, in dem Sie die ersten Daten des Files ablegen kann. Wurde einer gefunden, so wird seine Track und Sektornummer ebenfalls im Directoryeintrag gespeichert. Im andern Fall wird mit der Meldung "Disk Full" abgebrochen. Nun wird damit begonnen, die zu schreibenden Daten vom Rechner zu empfangen und zu speichern. Da ein Diskettenblock nur 256 Bytes lang ist, und zu speichernde Daten in der Regel ein größeres Volumen haben, sucht sich die Floppy jedesmal wieder einen neuen freien Block, wenn der letzte vollgeschrieben ist. Diese Blocks werden anschließend als 'belegt' gekennzeichnet und auf der Diskette vermerkt.
Nun fragen Sie sich bestimmt schon, wie das DOS später wieder erkennt, welche Blocks zu welchem File gehören und in welcher Reihenfolge diese geladen werden sollen. Das ist nun ganz einfach gelöst.
Wie ich oben schon erwähnte kann ein Datenblock 256 Bytes fassen. Als reine Datenbytes werden davon jedoch nur 254 genutzt. Die ersten zwei Bytes sind nämlich für die Folgeblocknummer reserviert. Im ersten Byte steht der Track, im zweiten der Sektor des nächsten Blocks, der zu diesem File gehört. In diesem ist dann wiederum der nächste Block vermerkt und so weiter. Um festzustellen, wann ein File aufhört, enthält der letzte Block als Tracknummer den Wert 0. Im Byte für die Sektornummer ist nun ein Wert enthalten, der die Anzahl der in diesem Block benutzten Bytes kennzeichnet, da am Ende, je nach Länge der gespeicherten Daten, mit hoher Wahrscheinlichkeit nicht alle Datenbytes des Blocks für das File benötigt werden.
Dieser Aufbau gilt übrigens für alle Filetypen. Ausnahmen bestätigen jedoch die Regel, und so ist es auch in diesem Fall. Relative Datenfiles benutzen eine andere Datenstruktur, was in Ihrer besonderen Funktion begründet ist. Sie benutzen sogenannte " Side–Sector–Blocks" um ihren relativen Aufbau verwalten zu können. Da die Manipulation an relativen Files jedoch wenig ratsam ist, und nur selten einen Sinn macht, wollen wir den Aufbau von Side-Sektor- Blöcken wegfallen lassen.

DIE DIRECTORY-BLOCKS

Nun wissen wir also, auf welche Weise Datenblocks miteinander verkettet sind. Was uns nun interessieren soll, ist die Art und Weise, mit der das DOS nun eine Diskette verwaltet. Im Prinzip haben wir das in obigem Beispiel auch schon angesprochen, nur wie funktioniert z. B. das Heraussuchen eines freien Directoryeintrags, oder freien Blocks?
Nun, wie schon erwähnt, ist der Track 18 ausschließlich dem DOS vorbehalten. Im Prinzip ist er komplett für das Directory zuständig, wobei hier natürlich auch Informationen enthalten sind, die wir nicht beim üblichen LOAD"$",8 angezeigt bekommen. Außerdem ist in einem ganz speziellen Block die sogenannte "BAM" abgelegt, was für "Block Availabillity Map" steht.

Übersetzt bedeutet das nichts anderes als "Block-Verfügbarkeits-Karte", bzw. Liste. Hier steht eingetragen, welche Blocks der Diskette schon belegt sind, und welche nicht.

DER DISK-HEADER-BLOCK

Den Block, in dem die BAM enthalten ist, nennt man "Disk-Header- Block". Er ist der wichtigste Block auf der ganzen Diskette, da nur an ihm das DOS eine Diskette überhaupt erkennt und bearbeiten kann. Er liegt im Sektor 0 des 18.
Tracks. Außer der BAM sind hier noch viele andere Informationen gespeichert, wie z. B. Diskettenname und ID. Wollen wir uns nun eine Liste anschauen, aus der Sie ersehen können, welche Bytes des Disk-Header- Blocks welche Informationen enthalten:

Byte	Aufgabe
000	Hier ist die Tracknummer des ersten Directoryblocks enthalten (normalerweise 18).
001	Hier steht die Sektornummer des ersten Directoryblocks (normalerweise 1).
002	Hier steht das 1541–Formatkennzeichen. Selbiges entspricht dem ASCII-Zeichen "A" (Code: 65 = $41)
003	Dieses Byte kennzeichnet, ob die eingelegte Diskette doppelseitig formatiert ist (dann steht hier der Wert 1). Das ist wichtig für die Benutzer einer 1571- Floppy, die in der Lage ist, Disketten beidseitig zu beschreiben. Die 1541 beachtet dieses Byte nicht.
004 – 007	In diesen 4 Bytes ist die Blockbelegung des ersten Tracks vermerkt.
008 – 139	Blockbelegungsbytes für die Tracks 2-34
140 – 143	Hier steht die Blockbelegung des letzten,35. Tracks
144 – 159	Hier steht der Diskettenname, der bei der Formatierung angegeben wurde, und zwar im ASCII- Code. Ist ein Name kürzer als 16 Zeichen, so werden die restlichen Bytes mit dem Wert 160 aufgefüllt.
160 – 161	Hier steht zweimal der Wert 160, was übrigens dem ASCII-Zeichen 'SHIFT-SPACE' entspricht.
162 – 163	Hier ist die zweistellige ID der Diskette vermerkt.
164	Wert 160
165 – 166	Formatangabe der Diskette. Diese ist bei 1541–Disketten immer "2A". Sie sehen sie im normalen Directory, daß Sie mit LOAD"$",8 geladen haben, immer am Ende der ersten Zeile, in der auch Diskname und ID zu finden sind.
167 – 170	Gefüllt mit 'SHIFT-SPACE' (Wert 160).
171 – 255	Unbenutzer Bereich. Gefüllt mit Nullen.

Wie Sie sehen kann hier schon eine Fülle an Informationen abgelesen werden. Zunächst einmal sehen wir hier, bei welchem Block das Directory beginnt. Das ist in der Regel Sektor 1 von Track 18. Dieser Wert könnte jedoch mit Hilfe eines Diskmonitors, oder eines kleinen Block–Lese– und Schreibprogramms beliebig geändert werden.
Das Formatkennzeichen ist bei der Floppy 1541 das "A". Es zeigt dem DOS an, daß dies eine von einer 1541 (oder einem kompatiblen Laufwerk, z. B.1570,1571) formatierte Diskette ist. Dieses Formatkennzeichen wurde deshalb eingeführt, weil Commodore für ältere Bürorechner ebenfalls eigene Laufwerke gebaut hat, die wiederum eine andere Diskettenstruktur aufweisen (z.B. mehr Tracks und Sektoren). Disketten von solchen Laufwerken können von der 1541 zwar teilweise gelesen, nicht aber beschrieben werden. Deshalb verweigert die Floppy auch den Schreibzugriff auf Disketten mit anderem Formatkennzeichen. Dies kann man sich aber auch zunutze machen: ändert man nämlich den Wert des 3. Bytes, so kann man eine Diskette softwaremäßig schreibschützen. Ein Speichern, Löschen, oder Softformatieren ist nun nicht mehr möglich. Wohl aber kann von der Diskette geladen werden. Das 4. Byte ist für die 1541

nutzlos. Hier steht in der Regel eine 0 (nur sinnvoll in Verwendung von Laufwerken mit 2 Schreib/ Leseköpfen).
Wie Sie weiterhin aus obiger Liste erkennen können, sind nun jeweils vier Bytes für die Blockbelegung eines jeden der 35 Tracks reserviert. Die Bytes 4 bis 143 enthalten die oben schon angesprochene BAM. Auf sie werden wir gleich zurückkommen. Es folgen zum Schluß noch zwei Informationen über die Diskette. Zum einen der 16 Zeichen lange Diskettenname, zum anderen die ID, gefolgt von einem 'SHIFT-SPACE' und der DOS-Versionsnummer (normalerweise "2A"). Auch diese beiden Bereiche können verändert werden. Zusätzlich sollten Sie wissen, daß Sie nicht nur die ID, sondern auch das SHIFT–SPACE– Zeichen, sowie das "2A" verändern können. Dadurch hat man die Möglichkeit eine 5–stellige "ID" herzustellen (natürlich zählen für das DOS immer nur noch die ersten beiden Zeichen). Auf diese Art und Weise ist es z.B. möglich, daß die Diskettenseiten der Magic-Disk die IDs "SIDE1" und "SIDE2" haben (haben Sie einmal darauf geachtet?).

DIE BAM

Die BAM verwaltet, wie schon erwähnt, die freien und belegten Blocks einer Diskette.140 Bytes sind für sie im Disk–Header–Block reserviert. Wir haben ebenso gelernt, daß jeweils vier Bytes die Blockbelegungen eines Tracks kodieren. Wollen wir uns dies einmal anhand des ersten Tracks betrachten. Dieser verfügt ja über 21 Sektoren. Seine "BAM-Bytes" sind in den Bytes 4 bis 7 des Disk–Header–Blocks (18/0) zu finden. Im ersten Byte dieser vier Bytes ist nun die Anzahl der freien Blöcke auf Track 1 vermerkt. In den folgenden drei Bytes sind nun die Zustände der Blocks 0–20 bitweise kodiert. Ist ein Bit in einem der drei Bytes gesetzt, so entspricht das der Information, daß der zugehörige Block frei ist. Ist es gelöscht, so ist der Block belegt. Die Verteilung der Sektoren auf die drei Bytes sieht nun folgendermaßen aus. Die Bits 7–0 (in dieser Reihenfolge!) des ersten Bytes geben die Belegung der Sektoren 0–7 an. Die Bits 7–0 des zweiten Bytes zeigen den Status für die Blocks 8–15 . Die Bits 7–3 des dritten Bytes stehen für die Blocks 16–20. Bits 2–0 sind hier unbelegt. Zur besseren Übersicht hier nochmal eine Tabelle ("DHB"= Disk–Header–Block):

Byte 1 (für Track 1, Byte 5 des DHB)

Bit	7	6	5	4	3	2	1	0
Sektor	00	01	02	03	04	05	06	07

Byte 2 (für Track 1, Byte 6 des DHB)

Bit	7	6	5	4	3	2	1	0
Sektor	08	09	10	11	12	13	14	15

Byte 3 (für Track 1, Byte 7 des DHB)

Bit	7	6	5	4	3	2	1	0
Sektor	16	17	18	19	20	--	--	--

Alle gesetzten Bits (= freier Block) dieser drei Bytes werden nun gezählt und in das 0. Byte der entsprechenden Track-BAM (für Track 1, Byte 4 des DHB), eingetragen. So baut sich nun die gesamte BAM auf, wobei sich in Byte 3 die Belegung natürlich ändert, da es ja auch Tracks gibt, die nur 19,18 oder 17 Sektoren haben. In letztem Fall sind dann nur noch das 7. und 6. Bit von Byte 3 benutzt.

DIE DIRECTORY-BLOCKS

Kommen wir nun zu den Directoryblocks. In ihnen sind die auf der Diskette enthaltenen Filenamen und deren Typ gespeichert. In einen Directoryblock passen 8 Fileeinträge. Sind alle Einträge voll, so wird ein neuer Directoryblock angelegt. Track und Sektor des erste Directoryblocks können aus dem Disk–Header–Block entnommen werden. Normalerweise ist das Block 18/1. Kommen wir nun zum Aufbau eines Directoryblocks. Zunächst einmal sind diese Blocks ebenso wie normale Datenblocks über die Angabe des jeweils nächsten Blocks miteinander verkettet. Deshalb stehen in den ersten beiden Bytes jeweils Track und Sektor des nächsten Directoryblocks. Der Letzte ist mit dem Wert 0 als Track und dem Wert 255 als Sektornummer markiert. Hier nun eine Übersicht mit der genauen Belegung eines Directoryblocks:

Byte Nummer	Aufgabe
000	Track des Folgeblocks
001	Sektor des Folgeblocks
002 – 033	Fileeintrag Nr. 1
034 – 065	Fileeintrag Nr. 2
066 – 097	Fileeintrag Nr. 3
098 – 129	Fileeintrag Nr. 4
130 – 161	Fileeintrag Nr. 5
162 – 193	Fileeintrag Nr. 6
194 – 225	Fileeintrag Nr. 7
226 – 255	Fileeintrag Nr. 8

Jeder einzelne Eintrag belegt also 32 Bytes im Directoryblock. Ich muß dies allerdings korrigieren. Im Prinzip werden immer nur die ersten 30 Bytes benutzt. Die letzten beiden Bytes eines Eintrags sind immer unbenutzt. Daher ist Eintrag Nr.8 auch nur 30 Bytes lang. Die 2 unbenutzten Bytes fehlen hier einfach, was auch keinen Unterschied macht. Nun wollen wir uns einmal anschauen, welche Informationen in den 30 Bytes eines File-Eintrags zu finden sind. Auch hier will ich Ihnen eine Tabelle geben, damit die Übersichtlichkeit gewahrt bleibt:

Byte Nummer	Aufgabe
000	Byte für den Filetyp
001 – 002	Track und Sektor des ersten Datenblocks dieses Files
003 – 018	16 Bytes für den Filenamen
019 – 020	Bei relativen Files stehen hier Track und Sektor des ersten Side–Sektor–Blocks (sonst unbenutzt).
021	Bei relativen Files seht hier die Länge eines Datensatzes (sonst unbenutzt).
022 – 025	unbenutzt
026 – 027	Zwischenspeicher des Tracks undektors beim Überschreiben mit dem Klammeraffen (" ") vor dem Filenamen. Dies ist nur ein temporärer Speicher. Normalerweise werden diese Bytes nicht beschrieben.
028 – 029	Hier steht die Länge des Files in Blocks als 16-Bit–Lo/Hi–Zahl.

In Byte 0 ist der Filetyp kodiert. Desweiteren finden sich hier auch Informationen darüber, ob das entsprechende File ordnungsgemäß geschlossen wurde und ob es schreibgeschützt ist. In ersterem Fall ist das File im normalen Directory mit einem "*" markiert. Das zeigt an, daß das

File ungültig ist. Ein File ist z. B. ungültig, wenn Sie versuchen etwas auf eine volle Diskette zu speichern. Die Floppy erkennt nun während des Schreibens, das kein Platz mehr vorhanden ist, und bricht ab. Zuletzt wird das fehlerhafte File noch als ungültig erklärt, damit man es nicht mehr laden kann. Mit einem Validate-Befehl an die Floppy wird ein solches File dann aus dem Directory entfernt. Ein schreibgeschütztes File kann nicht mehr überschrieben oder gelöscht werden, solange es geschützt ist. Dadurch können Sie das ungewollte Löschen von Daten verhindern. Ein so geschütztes File ist im Directory mit einer spitzen Klammer nach links ("<") markiert. Wenn Sie sich einmal das Inhaltsverzeichnis der Magic–Disk anschauen, so werden Sie dort mehrere Trennstrich–Files finden, die schreibgeschützt sind. Im Filetyp–Byte werden die einzelnen Informationen nun folgendermaßen codiert: Ist Bit 7 gesetzt, so ist ein File gültig, es wurde ordnungsgemäß geschlossen und kann geladen werden. Wenn Bit 6 gesetzt ist, so ist das File schreibgeschützt (normalerweise ist es gelöscht, also ungeschützt). Die Bits 0 bis 2 enthalten die möglichen Filecodierungen. Werte von 0 bis 4 sind sinnvoll. Deren Bedeutung ist folgendermaßen aufgeteilt:

Bits 2 – 0	Wert	Filetyp
000	0	DEL (deleted)
001	1	SEQ (sequentielle Datei)
010	2	PRG (Programmdatei)
011	3	USR (Userdatei)
100	4	REL (relative Datei)

Wie Sie sehen, können aber noch andere Werte eingetragen werden, die zwar keinen Filetyp spezifizieren, aber dennoch möglich sind. Dies hat zur Folge, daß die Files ganz merkwürdige Bezeichnungen erhalten. Wird zum Beispiel der Wert 15 (Bit 3 auch noch mitbenutzt) eingetragen, so hat ein File die Bezeichnung "?". Alle anderen Bits des Filetyp-Bytes eines Directoryeintrags sind unbenutzt. Ebenfalls interessant sind das 28 . und das 29. Byte eines Fileeintrags. Damit können Sie nämlich ebenfalls einen kleinen Effekt für den Directoryeintrag erzielen. Da hier die Länge des Files in Blocks angegeben ist, kann diese auch verändert werden. Dies hat keinerlei Einfluß auf das Laden oder Speichern eines Files, da die Blockangabe vom DOS nur dazu verwendet wird, um beim Ausgeben eines Directorys die Filelänge direkt greifbar zu haben. Wenn Sie nun jemanden verblüffen wollen, so kann hier z.B. der Wert 0 eingetragen werden. Laut Directory befindet sich in dem File dann nichts. Trotzdem kann sich dahinter ein Wahnsinns-Programm verstecken.
Einen ähnlichen Effekt können Sie auch durch Manipulation der BAM erzielen.Schreiben Sie hier nämlich andere Werte in die jeweils 0. Bytes der Trackbelegungen, so erhalten Sie eine andere Anzahl an freien Blocks. Das DOS addiert nämlich diese 35 Bytes auf, um die verbleibende Speicherkapazität ("Free Blocks") einer Diskette zu ermitteln und im Directory anzeigen zu können. Achten Sie danach jedoch bitte darauf, daß Sie nichts mehr auf eine solche Diskette speichern, da nun nämlich vermeintliche freie Blöcke beschrieben werden könnten, was zu erheblichem Datenverlust führen kann. Am Besten ist es dann, einen Schreibzugriff durch Ändern des Formatkennzeichens zu verhindern. Übrigens sollte ich erwähnen, daß Änderungen an der BAM jederzeit durch den Validate-Befehl der Floppy wieder rückgängig gemacht werden können. Dieses Floppykommando geht nämlich nach und nach alle Files der Diskette durch und merkt sich die jeweils belegten Blocks. Hiernach kann die ursprüngliche BAM wieder rekonstruiert werden.
Und noch etwas ist zu den Directoryeinträgen zu sagen. Wenn Sie ein File löschen, so wird es keineswegs komplett von der Diskette entfernt. Das einzige, was der Scratch-Befehl bewirkt, ist das Freigeben der belegten Diskettenblocks. Desweiteren wird als Filetyp der Wert 0 eingetragen, was dem Filetyp "DEL" entspricht. Gleichzeitig ist das File als 'ungültig' markiert und nicht schreibgeschützt (wieso auch). Ein solcher Fileeintrag kann jederzeit

wiederhergestellt werden, indem man einen definierten, gültigen, Filetyp einträgt und anschließend die Diskette validiert. Nun ist das File wieder so vorhanden, wie vor dem Löschen (inklusive des Vermerks, welche Blocks belegt sind). Wird jedoch zwischenzeitlich etwas auf der Diskette gespeichert, so sucht sich das DOS zunächst immer einen solchen 'leeren' Fileeintrag heraus, bevor es einen neuen Eintrag im letzten Directoryblock anlegt. Dadurch erklärt sich auch, warum neu geschriebene Files manchmal mitten im Directory zu finden sind und nicht am Ende. Hier wurde vorher an dieser Stelle ein altes File gelöscht.
Dies soll es dann wieder einmal gewesen sein für diesen Monat. Wenn Sie Ihre neu hinzugewonnenen Kenntnisse einmal ausprobieren möchten, so nehmen Sie sich einfach einen Diskmonitor zur Hand und versuchen Sie einmal einige Directoryeinträge oder den Disk-Header- Block zu verändern. Weiterhin kann mit dem Programm "Disk-Manager" auf dieser MD das Directory noch weitaus komfortabler manipuliert werden. Nun wissen Sie ja wahrscheinlich auch, was dabei geschieht.
Im nächsten Teil des Floppykurses wollen wir uns mit der Floppybedienung in Assembler beschäftigen. Dabei werden wir dann auch einige Programme erstellen, die sich die hier kennengelernten Informationen zunutze machen.

(ub)

Teil 5 – Magic Disk 10/92

Herzlich Willkommen zum 5. Teil unseres Floppykurses. In diesem Monat wollen wir die Kenntnisse, die wir bisher über die Floppy erfahren haben in die Praxis umsetzen. Das heißt, daß wir in diesem Kursteil einige nützliche Anwendungen zum bisher Erlernten programmieren werden. Diese werden zunächst in BASIC behandelt. Im nächsten Kursteil wollen wir dann auf die Programmierung in Assembler eingehen.

1. DISKETTE SCHREIBSCHÜTZEN

Kommen wir zu unserem ersten Programmbeispiel. Wie ich im dritten Teil dieses Kurses schon erwähnte ist es ganz einfach möglich, eine Diskette softwaremäßig schreib zuschützen. Sie können anschließend weder etwas auf die Diskette schreiben, noch etwas von ihr löschen.Das Laden von Programmen funktioniert jedoch nach wie vor. Einzige Möglichkeit die Diskette wieder beschreibbar zu machen ist dann das Formatieren (oder nochmaliges Behandeln mit unserem Programm).
Wie realisieren wir nun einen solchen Schreibschutz? Nun, wie wir mittlerweile ja wissen, legt die Floppy beim Formatieren einer Diskette im Diskheaderblock einige wichtige Informationen über die Diskette ab. Hier steht unter anderem auch das Formatkennzeichen, das bei der 1541 den Buchstaben "A" darstellt. Andere Commodorelaufwerke (z.B. das eines alten CBM4040) haben dort ein anderes Zeichen stehen. Da Commodore bei den alten Laufwerkstypen ähnliche Aufzeichnugsformate benutzte, sind die Disketten von Commodorelaufwerken bis zu gewissen Grenzen untereinander kompatibel. Meist unterscheiden sie sich nur in der Aufzeichnungsdichte (mehr Tracks auf einer Diskette). Um nun zu vermeiden, das Laufwerke von verschiedenen Rechnern die Daten einer Diskette eines anderen Laufwerks zerstören, wurde eine Sicherung in die 1541 eingebaut. Sie beschreibt ausschließlich nur Disketten, die ein "A" als Formatkennzeichen tragen. Andere werden nicht verändert, wohl aber kann von Ihnen gelesen werden. Es genügt nun also, das Formatkennzeichen in ein anderes Zeichen umzuwandeln, und die 1541 wird keine Daten mehr darauf schreiben. Dies geschieht im Prinzip ganz einfach. Wir müssen lediglich den Diskheaderblock einlesen, das 3. Byte (hier steht nämlich das Formatkennzeichen) in ein anderes Zeichen als "A" abwandeln und den Block wieder auf die Diskette zurückschreiben. Abschließend muß die Diskette noch initialisiert werden, damit das Betriebssystem der Floppy, das DOS, die neue Formatkennung auch in den Floppyspeicher übernimmt.

Ein Programm, das eine Diskette mit Schreibschutz versieht und ihn wieder entfernt finden Sie auf dieser MD unter dem Namen "FK.SCHREIBSCHUTZ". Die Unterroutine, mit der eine Diskette geschützt wird, steht in den Zeilen 400 bis 470 dieses Programms. Hier sind sie nochmals aufgelistet:

Unterroutine "Diskette sichern":

```
400  OPEN 1,8,15,"I": OPEN 2,8,2,"#"
410  PRINT#1,"U1 2 0 18 0"
430  PRINT#1,"B-P 2 2"
440  PRINT#2,"B";
450  PRINT#1,"U2 2 0 18 0"
460  PRINT#1,"I"
470  CLOSE1:CLOSE2:RETURN
```

Zunächst einmal öffnen wir hier den Befehlskanal und übergeben gleichzeitig einen Initialisierungsbefehl an ihn. Mit letzterem werden alle Grunddaten der eingelegten Diskette (z.B. ID, Formatkennung, etc.) in den Speicher der Floppy übertragen. Man sollte vor einer direkten Manipulation des Diskheaderblocks diesen Befehl aufrufen. Desweiteren öffnen wir einen Pufferkanal mit der Kanalnummer 2 (also Sekundäradresse 2) um Blockoperationen durchführen zu können. In Zeile 410 wird nun Spur 0 von Track 18 in den Puffer, dem Kanal 2 zugeordnet ist, eingelesen. Dies tun wir mit dem "U1"–Befehl, der besser ist als "Block–Read" ("B–R"). Hiernach folgt die Positionierung des Pufferzeigers auf das 2. Byte (mit dem 0. Byte eigentlich das dritte Byte) des Puffers. In Zeile 440 wird nun der Buchstabe "B" an diese Position geschrieben. Wichtig ist hierbei das Semikolon (";") nach dem PRINT-Befehl, da letzterer ohne Semikolon noch ein CHR$(13) nachsenden würde, womit wir einen Teil der BAM überschreiben würden! Nun muß der geänderte Block nur noch mit Hilfe des "U2"–Befehls auf die Diskette zurückgeschrieben werden. Damit unsere Änderung auch in den Floppyspeicher übernommen wird muß abschließend die Diskette wieder initialisiert werden. In Zeile 470 werden die beiden Filekanäle wieder geschlossen und zum Hauptprogramm zurückgekehrt.
Nun soll unser Programm den Diskettenschutz ja auch wieder rückgängig machen können. Dies geschieht im Prinzip wieder genauso. Wir müssen lediglich die fremde Formatkennung wieder in ein "A" umwandeln. Jedoch ergibt sich hierbei ein kleineres Problem. Dadurch, daß die Diskette ja schreibgeschützt ist, können wir den modifizierten Disk-Header ja nicht mehr auf die Diskette zurückschreiben! Hier behilft man sich mit einem kleinen Trick. In der Speicherstelle $0101(dez.257) des Floppy–RAMs "merkt" sich das DOS das Formatkennzeichen der eingelegten Diskette. Es wird bei jedem Initialisierungsbefehl wieder aktualisiert. Ändern wir nun diese Speicherstelle so um, daß in ihr der Wert "A" steht, so können wir dem DOS vorgaukeln, es hätte eine Diskette mit korrekter Formatkennung vor sich. Erst jetzt ist es wieder möglich auf die Diskette zu schreiben. Auch hier möchte ich Ihnen die Unterroutine, die entsprechendes tut auflisten. Sie steht im oben erwähnten Programm in den Zeilen 300–370 :

Unterroutine "Diskette entsichern":

```
300  OPEN 1,8,15,"I": OPEN 2,8,2,"#"
310  PRINT#1,"U1 2 0 18 0"
320  PRINT#1,"M-W";CHR$(1);CHR$(1);CHR$(1);CHR$(65);
330  PRINT#1,"B-P 2 2"
340  PRINT#2,"A";
350  PRINT#1,"U2 2 0 18 0"
360  PRINT#1,"I"
370  CLOSE1:CLOSE2:RETURN
```

In den Zeilen 300 und 310 öffnen wir, wie gewohnt, unsere beiden Filekanäle und lesen den Diskheaderblock in den Puffer ein. Anschließend jedoch benutzen wir den Memory-Write- befehl der Floppy um den Inhalt von $0101 zu in "A" abzuändern. Die Adresse $0101 wird durch das

Low–Byte 1 und das High–Byte 1 dargestellt. Desweiteren wollen wir nur ein Byte übertragen. Selbiges hat den ASCII–Code 65, was dem Buchstaben "A" entspricht. In dieser Reihenfolge werden die Parameter nun an den String "M–W" angehängt. Nach dieser Operation können wir nun den Buffer-Pointer auf das Formatkennzeichen positionieren und über den Pufferkanal den Buchstaben "A" reinschreiben. Da die Floppy seit dem "M–W"–Befehl glaubt, es läge eine Diskette mit korrekter Formatkennung vor, können wir in Zeile 350 auch erfolgreich den Diskheaderblock zurückschreiben. Eine Abschließende Initialisierung macht die Diskette ab nun wieder beschreibbar. Nun soll unser Programm aber auch in der Lage sein, zu erkennen, ob eine Diskette schreibgeschützt ist, oder nicht. Auch dies ist sehr einfach zu realisieren. Wir gehen dabei wieder den Weg über die Speicherstelle $0101 des Floppy-RAMs.
Hier die Unterroutine die in den Zeilen 200 bis 260 zu finden ist:

Unterroutine "Formatkennung prüfen:"

```
200  OPEN 1,8,15,"I"
210  PRINT#1,"M–R";CHR$(1);CHR$(1);CHR$(1)
220  GET#1,A$
230  PRINT "DISKETTE IST ";
240  IF A$="A" THEN PRINT "NICHT";
250  PRINT "GESCHUETZT!"
260  CLOSE1:RETURN
```

Hier wird zunächst nur der Befehlskanal geöffnet. Einen Puffer benötigen wir nicht. Gleichzeitig wird die Diskette initialisiert und somit ihr Formatkennzeichen in die Speicherstelle $0101 übertragen. In Zeile 210 greifen wir diesmal mit Hilfe des Memory-Read- Befehls ("M–R") auf selbige zu. Wieder werden Low- und High-Byte 1, sowie die Länge 1 als CHR$–Codes übergeben. Anschließend können wir das Zeichen aus dem Befehlskanal auslesen. Es folgt nun eine Prüfung, ob es dem Buchstaben "A" entspricht. Wenn ja, so wird zuerst der Text "nicht" ausgegeben und anschließend der Text "geschuetzt!". Andernfalls gibt das Programm nur " geschuetzt" aus. Zum Schluß schließen wir den Befehlskanal wieder und kehren zum Hauptprogramm zurück.

2. ÄNDERN DES DISKETTENNAMENS UND DER ID

Sicherlich haben Sie schon einmal ein Programm benutzt, das den Namen und die ID einer Diskette nachträglich verändert, ohne Daten zu zerstören. Wollen wir nun einmal selbst so etwas schreiben.
Zunächst möchten wir den Diskettennamen verändern. Wie Sie aus dem letzten Kurs bestimmt noch wissen, steht der Name einer Diskette in den Bytes 144–159 des Dir–Header–Blocks. Um ihn zu ändern brauchen wir lediglich diesen Block in einen Pufferspeicher einzulesen und den neuen Namen an dieser Position hineinzuschreiben. Dabei müssen wir noch auf zwei Dinge achten: Ist der neue Name kürzer als 16 Zeichen, so müssen die restlichen Zeichen mit dem ASCII–Code 160 aufgefüllt werden. Das ist der Code für 'SHIFT–SPACE' der von der Floppy automatisch an kürzere Namen angefügt wird. Desweiteren darf der neue Name natürlich nicht länger als 16 Zeichen sein, da wir sonst ja weitere Daten im Diskheaderblock überschreiben würden. Das Programm mit dem Sie Disknamen und -ID ändern können, finden Sie auf dieser MD unter dem Namen "FK.NAMECHANGE". Hier nun ein Auszug mit der Namensänderungsroutine bei Zeile 200:

Unterroutine " Diskname ändern" :

```
200  OPEN 1,8,15,"I":OPEN 2,8,2,"#"
210  PRINT#1,"U1 2 0 18 0"
220  PRINT#1,"B-P 2 144"
230  NA$=""
240  FOR I=1 TO 16
250  GET#1,A$: NA$=NA$+A$
```

```
260  NEXT
270  PRINT "AKTUELLER NAME:";NA$
280  INPUT "NEUER NAME";NA$
285  IF LEN(NA$)>16 THEN NA$=LEFT$(NA$,16)
290  IF LEN(NA$)<16 THEN NA$=NA$+CHR$(160):GOTO290
300  PRINT#1,"B-P 2 144"
310  PRINT#2,NA$;
320  PRINT#1,"U2 2 0 18 0"
330  PRINT#1,"I"
340  CLOSE1:CLOSE2:RETURN
```

In den Zeilen 200 und 210 öffnen wir, wie gewohnt, unsere Kanäle und lesen den Diskheader–block in den Floppypuffer. Anschließend wird der Pufferzeiger auf das 144. Byte des Puffers positioniert. In den Zeilen 230-260 werden nun die 16 Bytes des Diskettennamens mit Hilfe einer Schleife ausgelesen und in der Variablen NA$ abgelegt. Diese wird anschließend ausgegeben um den aktuellen Diskettennamen anzuzeigen. Es wird daraufhin nach dem neuen Namen gefragt, der nun seinerseits in die Variable NA$ kommt. In der Zeile 285 wird zunächst geprüft, ob der neue Name zu lang ist. Wenn ja, so wird er auf die ersten 16 Zeichen gekürzt. In Zeile 290 wird abgefragt, ob der eingegebene Name nicht zu kurz ist. Wäre das der Fall, so würde beim Schreibvorgang der alte Name nicht ganz überschrieben werden und wäre dann noch sichtbar. In diesem Fall füllt die Schleife in Zeile 290 den Rest des Strings NA$ mit 'SHIFT-SPACES' auf. Erst jetzt kann wieder auf das erste Byte des Namens (Byte 144 des Diskheaderblocks) positioniert werden und der String NA$ dorthin geschrieben werden. Abschließend wird die Änderung mittels "U2" auf der Diskette aktualisiert, letztere nochmals initialisiert und alle Kanäle werden geschlossen. Wollen wir die ID einer Diskette verändern, so müssen wir ähnlich arbeiten. Hierbei wollen wir berücksichtigen, daß wir für die Directoryanzeige sogar fünf Zeichen (anstelle von 2) zur Verfügung haben. Sie sicherlich ist Ihnen aufgefallen, daß in den letzten drei Zeichen der ID im Directory (rechts oben) immer der Text "2A" steht, was eine Formatkennung darstellt. Diese Formatkennung hat jedoch keinerlei Einfluß auf die, die im dritten Byte des Diskheaderblocks zu finden ist. Sie wird lediglich zur Anzeige beim Laden des Diretorys benötigt und kann im Prinzip beliebig geändert werden. Die drei Formatkennzeichenbytes liegen glücklicherweise auch direkt hinter den beiden Bytes für die Disketten–ID, nämlich in den Bytes 162 – 166 des Diskheaderblocks. Hier nun der Programmteil, der die ID einer Diskette ändert:

Unterroutine "ID ändern"

```
400  OPEN 1,8,15,"I": OPEN 2,8,2,"#"
410  PRINT#1,"U1 2 0 18 0"
420  PRINT#1,"B-P 2 162"
430  ID$=""
440  FOR I=1 TO 5
450  GET#1,A$: ID$=ID$+A$
460  NEXT
470  PRINT "AKTUELLE ID:";ID$
480  INPUT "    NEUE ID";ID$
490  IF LEN(ID$)>5 THEN ID$=LEFT$(ID$,5)
500  PRINT#1,"B-P 2 162"
510  PRINT#2,ID$;
520  PRINT#1,"U2 2 0 18 0"
530  PRINT#1,"I"
540  CLOSE1: CLOSE2: RETURN
```

Im Prinzip arbeitet diese Unterroutine genauso wie die, die den Namen ändert. Nur daß diesmal nicht 16, sondern nur 5 Zeichen für die ID eingelesen werden müssen. Desweiteren kann eine ID auch kürzer sein, als 5 Zeichen, nämlich dann, wenn Sie wirklich nur die beiden ID–Zeichen

ändern wollen, das Formatkennzeichen aber gleich bleiben soll. Deshalb wird in Zeile 490 der String ID$ nur auf 5 Zeichen gekürzt, falls er länger sein sollte. Ein kürzerer Text wird als solcher übernommen und an die Entsprechende Position (nämlich 162) geschrieben. Auch hier wird die Änderung dann mittels "U2" gespeichert und die Diskette initialisiert.
Das soll es dann wieder einmal gewesen sein für diesen Monat. Ich hoffe ich habe Sie zum " Spielen" mit den Blockbefehlen animiert. Nächsten Monat wollen wir uns mit einer weiteren Anwendung der Blockbefehle beschäftigen und dann die Programmierung der Floppy in Maschinensprache angehen.

(ub)

Teil 6 – Magic Disk 11/92

Hallo und herzlich Willkommen zum sechsten Teil unseres Floppykurses. Wir wollen unsere Kenntnisse auch hier wieder vertiefen und zwei weitere Beispielprogramme kennenlernen. Eines, um das Directory auf dem Bildschirm anzuzeigen und eines um gelöschte Files wieder zu retten.

BEISPIEL 1: **DIRECTORY ANZEIGEN**
Unser erstes Programmbeispiel soll eine Routine sein, mit der Sie das Directory der eingelegten Diskette ohne Speicherverlust einlesen können. Dies ist ein Hinweis auf eine besondere Eigenheit der Floppy. Wenn wir uns nämlich das Inhaltsverzeichnis einer Diskette einfach nur anzeigen lassen wollen, so können wir durch Angabe des Filenamens "$" die Floppy dazu bewegen, uns sehr viel Arbeit abzunehmen. Sie beginnt dann nämlich damit, alle Directoryblöcke von selbst einzulesen und sie für uns aufzubereiten. Ein direktes "herumpfuschen" auf der Diskette mittels der Block-Befehle entfällt.
Wollen wir uns nun verdeutlichen, was passiert, wenn wir wie gewohnt das Directory mit LOAD"$",8 in den Speicher des Computers laden. Wie schon erwähnt, ist das Dollarzeichen für die Floppy das Kennzeichen für eine spezielle Aufbereitung des Disketteninhalts. Sie sendet an den C64 nun Daten, die dieser, wie bei auch beim Laden eines normalen Programms in seinen Basic-Speicher schreibt. Der Computer behandelt das Directory im Prinzip also wie ein Basic–Programm. Dies wird dadurch bestätigt, daß das eingelesene Directory auch wie ein Basic-Programm, mit dem Befehl LIST auf dem Bildschirm angezeigt wird. Sie können es sogar mit NEW wieder löschen, oder mit RUN starten, was jedoch nur einen Syntax–Error bewirkt, da das Directory ja keine echten Basic–Befehle enthält. Sinnigerweise übermittelt die Floppy in der Directoryliste als erstes auch immer die Blockanzahl eines Files, was für das Basic des C64 einer Zeilennummer entspricht. Diese Zeilennummern sind zwar nicht immer in einer numerischen Reihenfolge, jedoch macht das Basic unseres Computers da keinen Unterschied. Der interne Aufbau eines Basic–Programms erlaubt es tatsächlich, Zeilennummern wahllos zu vergeben. Wenn wir ein Programm direkt eingeben ist das natürlich nicht möglich, da die Eingaberoutine des Betriebssystems die Zeile anhand der Zeilennummer automatisch an der logisch richtigen Position des Programms einfügt.
Der langen Rede kurzer Sinn ist, daß wir beim Übermitteln des Directorys von der Floppy darauf achten müssen, daß diese es uns in einer Form schickt, die eigentlich nur der Basic–Programm- Speicher versteht, so daß die LIST-Routine das 'Directory-Programm' listen kann. Deshalb gibt es einige Bytes, die für uns wertlos sind und überlesen werden müssen. Hierzu erkläre ich Ihnen einfach einmal den Aufbau des Files, das uns die Floppy bei Angabe des Filenamens "$" übermittelt. Zuerst erhalten wir von ihr, wie bei jedem File, daß normalerweise mittels LOAD in den Speicher des Computers gelangt, eine Adresse, an die das "Programm" geladen werden soll. Dies ist die Basic-Startadresse $0801(dez. 2049) im Low–/ High–Byteformat. Wir benötigen diese natürlich nicht, und können sie deshalb überlesen. Von nun an ist der Aufbau einer jeden Directoryzeile gleich. Zuerst werden dabei zwei Linkbytes übertragen,

die der Basic–Interpreter benötigt, um die einzelnen Zeilen zu verketten. Beide können wir getrost überlesen. Als nächstes erhalten wir dann die Blockanzahl in Low–/ High–Byte–Darstellung, was für den Basic–Interpreter einer Zeilennummer entspräche. Diese Nummer müssen wir nun erst einmal umrechnen, bevor wir sie ausgeben können. Hieran anschließend folgen nun ein oder mehrere Spacezeichen, sowie der Filename und die Filekennung im Klartext. Alle diese letzten Zeichen werden im ASCII–Code übertragen, so daß wir sie lediglich mit PRINT ausgeben müssen. Am Ende bekommen wir dann noch eine Null übermittelt, die das Ende der (Basic–) Zeile kennzeichnet. An ihrer Stelle müssen wir lediglich einen Zeilenvorschub ausgeben (ein PRINT ohne Parameter). Das Ende des Files ist natürlich dann erreicht, wenn die Statusvariable ST gleich 64 ist. Im Prinzip ist alles also ganz einfach. Hier ist das entsprechende Basic-Programm, Sie finden es auf dieser MD unter dem Namen "FK.DIR":

```
 10 gosub200
 20 end
 30 :
 31 :
 90 rem *****************
 91 rem *zeichen lesen*
 92 rem *****************
 93 :
100 get#1,a$
110 ifa$=""thena=0:return
120 a=asc(a$):return
130 :
131 :
190 rem **********************
191 rem *directory anzeigen*
192 rem **********************
193 :
200 print"{clr}";:open1,8,0,"$"
210 fori=1to4:get#1,a$:next
215 :
220 gosub100:bl=a
230 gosub100:bl=bl+256*a
240 print bl;
250 get#1,a$:printa$;
260 ifa$<>""then250
270 print
280 fori=1to2:get#1,a$:next
290 ifst=0then220
300 close1
310 return
```

In den Zeilen 100 – 120 sehen Sie eine Routine, die uns ein Byte mittels GET# einliest und es als numerischen Wert in der Variablen "a" abspeichert. Diese Routine ist deshalb notwendig, da eine Umwandlung des Strings "a$" bei dem Bytewert 0 einen "illegal quantity error" verursacht. Das liegt daran, daß ein String, der nur den Bytewert 0 als Zeichen enthält, einem Leerstring entspricht. Deshalb muß mit der IF–THEN–Abfrage überprüft werden, ob "a$" ein Leerstring ist, oder nicht. Ab Zeile 200 sehen wir nun das Programm, das das Directory einliest und auf dem Bildschirm ausgibt. In Zeile 200 wird zunächst einmal der Bildschirm gelöscht und der Filekanal, der uns das Directory sendet, geöffnet. Ich habe hier ganz bewußt die Sekundäradresse 0 benutzt, da sie der Floppy ja automatisch mitteilt, daß wir ein Lesefile öffnen (siehe vorherige Teile dieses Kurses). In Zeile 210 werden nun die Startadresse und die beiden Linkbytes der ersten Zeile überlesen. Danach beginnt die Hauptschleife des Unterprogramms. In den Zeilen 220 und 230 lesen wir hier zunächst mit Hilfe unserer "Zeichen lesen"-Unterroutine, Low- und High-Byte der Blockanzahl aus, verrechnen diese beiden Werte zu der reellen Blockanzahl und

speichern sie in der Variablen "bl" (Wert=Lowbyte+256*–Highbyte). Die Blockanzahl wird jetzt in Zeile 240 auf dem Bildschirm ausgegeben.
Wir hängen an den PRINT–Befehl ganz bewußt ein Semikolon an, damit der folgende Text direkt hinter der Blockanzahl ausgegeben wird. Dies geschieht in den Zeilen 250 und 260 . Dort lesen wir jeweils ein Zeichen mittels GET# ein, und geben es anschließend auf dem Bildschirm aus. War das eingelesene Zeichen ein Leerstring, entspricht es dem Bytewert 0 und die Zeile ist zu Ende. Jetzt wird in Zeile 270 eine Leerzeile ausgegeben, daß der Cursor am Anfang der nächsten Zeile steht. In Zeile 280 werden die beiden Linkbytes der folgenden Directory–Zeile überlesen. In der IF–THEN–Abfrage in Zeile 290 wird überprüft, ob das Ende des Files schon überlesen wurde. Wenn nicht, wird an den Anfang der Hauptschleife verzweigt und alles beginnt von vorne. Wenn doch, so wird der Filekanal geschlossen und zum aufrufenden Programm zurückverzweigt.

BEISPIEL 2: **GELÖSCHTE FILES RETTEN**
Um Sie in die Benutzung der Blockbefehle weiter einzuführen, wollen wir uns jetzt einem weiteren Programmbeispiel in BASIC zuwenden. Wir wollen ein einfaches "UN–DELETE"–Programm schreiben. Mit ihm soll es möglich sein, versehentlich mit dem Floppybefehl "SCRATCH"("S:") gelöschte Dateien wiederherzustellen. Hierzu wollen wir zunächst einmal klären, was die Floppy eigentlich macht, wenn sie ein File löschen soll. Als erstes bekommt Sie einen Befehl übermittelt, der sie dazu auffordert ein (oder auch mehrere) Files zu löschen. Als Beispiel wollen wir einmal ein File mit Namen "TEST" heranziehen. Der entsprechende Floppybefehl an den Befehlskanal muß also lauten "S:TEST". Hieraufhin beginnt die Floppy nun, das Inhaltsverzeichnis der eingelegten Diskette nach einer Datei zu durchsuchen, die den Namen "TEST" trägt. Wird sie gefunden, so trägt die Lösch-Routine des Floppy–Betriebs–systems lediglich den Wert 0 als Filetyp für dieses File ein, und sucht nun alle Blocks heraus, die von dem File belegt werden. Jetzt holt sich die Floppy die BAM der Diskette in den Speicher, kennzeichnet alle vom File belegten Datenblocks als 'unbelegt' und schreibt die BAM wieder zurück auf die Diskette. Der Filetyp 0 entspricht dem Filetyp "DEL", der normalerweise im Directory nicht mehr angezeigt wird. An dieser Leerstelle wird das File einfach übersprungen. Im Prinzip sind aber noch alle seine Informationen auf der Diskette enthalten. Zumindest einmal solange, bis Sie wieder ein neues File auf die Diskette schreiben. Dann nämlich sucht sich die Floppy den ersten freien Directoryeintrag heraus, der in unserem Fall, der des Files "TEST" ist, und trägt dort das neue File ein. Desweiteren kann es dann auch passieren, daß die Datenblocks, die von "TEST" belegt wurden möglicherweise von denen des neuen Files überschrieben werden. Ein UNDELETE-Programm hat deshalb nur dann einen Sinn, wenn noch nichts neues auf die Diskette geschrieben wurde (obwohl es auch andere Programme gibt, die selbst dann noch das eine oder andere retten können – hierauf kommen wir später zurück). Es muß lediglich alle Einträge im Directory nach Filetyp-0-Einträgen durchsuchen und diese Anzeigen. Hieraufhin muß vom Benutzer entschieden werden, welchen Filetyp das alte File hatte (dies ist die einzige Information, die über selbiges verloren ging). Ist dieser angegeben, so braucht unser Programm nur noch den Entsprechenden Code in den Fileeintrag zu schreiben und alle Blocks des Files zu verfolgen, die von ihm belegt wurden und sie abermals als 'belegt' zu kennzeichnen. Dies kann man über die umständliche Metode tun, indem man aus den Bytes 1 und 2 des Fileeintrags Track und Sektor ersten Datenblocks ermittelt, und nun alle verketteten Blöcke (nächster Block steht immer in den ersten beiden Bytes eines Datenblocks) heraussucht uns diese mittels Block-Allocate-Befehl ("B-A") als belegt kennzeichnet. Die zweite Methode ist einfach den Validate-Befehl der Floppy zu benutzen. Dieser sucht nämlich automatisch alle Blocks von gültigen Fileeinträgen im Directory heraus und kennzeichnet sie als 'belegt'. Beide Methoden haben ihre Vor- und Nachteile.
Benutzen wir die erste Methode, so haben wir bei höherem Programmieraufwand eine weitaus höhere Arbeitsgeschwindigkeit (da nur die Blocks eines Files herausgesucht werden müssen und nicht aller Files der Diskette, was sich vor allem dann bemerkbar macht, wenn Sie

besonders viele Files auf der Diskette haben). Andererseits ist es bei dieser Methode nicht so einfach, die Blocks eines REL-Files wiederzubelegen, da diese mit zusätzlichen Side-Sektor-Blöcken arbeiten, die ebenfalls gesucht werden müssen. Das aber wiederum macht der Validate-Befehl für uns. Außerdem arbeitet er bei Disketten mit wenigen Files weitaus schneller als unsere Routine, da er ja ein Floppyprogramm ist und ein zeitraubender Datenaustausch zwischen Floppy und C64 entfällt. Da wir unser Programm in BASIC schreiben wollen, ist er bestimmt immer noch etwas schneller. Ich werde mich in meinem Beispiel nun jedoch auf die "von Hand"-Methode beschränken.
Wenn Sie möchten, können Sie das Programm ja so erweitern, daß z.B. bei weniger als fünf Files automatisch der Validate-Befehl benutzt wird und im anderen Fall die "von Hand"-Routine.
Zunächst will ich Ihnen nun eine Unterroutine vorstellen, die uns das Inhaltsverzeichnis einliest und alle Informationen darüber in Variablenfeldern ablegt. Sie steht in dem Beispielprogramm "FK.UNDEL" auf dieser MD in den Zeilen 200 – 510. Wir benötigen sie, um die einzelnen Fileeinträge zu isolieren und feststellen zu können, welcher Eintrag einem DEL-File entspricht. Ich habe die Routine jedoch so ausgelegt, daß Sie sie auch für andere Zwecke benutzen können, da sie so ziemlich alle Informationen des Directorys ausliest.
Vorher sind jedoch noch einige Anmerkungen zu der Routine zu machen. Vorab sei gesagt, daß in den Zeilen 600 – 620 dieselbe Routine steht, wie wir sie auch schon im Dir-Programm benutzten. Sie liest ein Zeichen ein, und wandelt es in einen Bytewert um, der nach Aufruf in der Variablen "a" steht. Desweiteren benötigen wir noch einige Variablenfelder, die im Hauptprogramm mit Hilfe der DIM-Anweisung dimensioniert werden müssen. Hier eine Liste der Felder:

Name	Element	Inhalt
TY$	144	Starttrack des Files
FT	144	Startsektor des Files
FS	144	Startsektor des Files
BL	144	Blockanzahl des Files
NA$	144	Name des Files
DT	18	Tracknummern des Dirs
DS	18	Sektorennummern des Dirs

Die ersten fünf Felder aus dieser Liste dienen ausschließlich der Speicherung von Informationen über ein File. Da das Directory maximal 144 Einträge beinhalten kann, werden alle diese Felder auf maximal 144 Elemente dimensioniert. Die Felder DT und DS werden benutzt um Track und Sektor der Folgeblocks des Directorys zu speichern. Rein theoretisch ist das zwar nicht unbedingt notwendig, da die Folge der Directoryblöcke immer gleich ist (Block 1 in 18/1, Block 2 in 18/4, etc.), jedoch ist so einfacher mit den Blocks zu handhaben. Im Prinzip könnten wir auch das Feld DT wegfallen lassen, da das Directory IMMER in Track 18 steht, jedoch kann es sich ja auch um eine manipulierte Diskette handeln, die das Directory woanders stehen hat (hierauf wollen wir in einem der nächsten Teile des Floppykurses zurückkommen).
Nun möchte ich Ihnen die Routine, mit denen wir die Diretcoryinformationen einlesen nicht länger vorenthalten. Hier der erste Teil von Zeile 200 bis 295:

```
200  print"Gelesene Files:"
210  open1,8,15,"i":open2,8,2,"#"
220  print#1,"u1 2 0 18 0"
230  gosub600:tr=a
240  gosub600:se=a
245  :
250  dn$="":id$=""
260  print#1,"b-p 2 143"
```

```
270 fori=0to15:get#2,a$:dn$=dn$+a$:next
280 print#1,"b-p 2 162"
290 fori=1to5:get#2,a$:id$=id$+a$:next
295 :
```

In diesem Teil unserer Unterroutine werden die Vorbereitungen zum Einlesen des Directorys getroffen, sowie der Diskettenname und die ID in die Variablen "dn$" und "id$" eingelesen. Diese Variablen können Sie im Hauptprogramm ebenfalls weiterverwenden. Nachdem also in Zeile 200 ein Informationstext ausgegeben wurde, öffnen wir in Zeile 210 den Befehlskanal und einen Pufferkanal. Ersterer erhält die Filenummer 1, letzterer die Filenummer 2. Beim Öffnen des Befehlskanals initialisieren wir die Diskette auch gleichzeitig. In Zeile 220 senden wir nun den ersten Befehl an die Floppy. Der U1-Befehl in dieser Zeile liest uns den Dir-Header-Block (Track 18, Sektor 0) in den Puffer. In Zeile 230 und 240 lesen wir nun die ersten beiden Bytes dieses Blocks ein, und ordnen sie den Variablen TR und SE zu, die als Variablen für den aktuellen Track/ Sektor benutzt werden. Sie enthalten nun Track und Sektor des ersten Directoryblocks. Bevor wir uns jedoch daran machen dieses auszulesen, nehmen wir uns aus dem Dir-Header-Block noch den Diskettennamen und ID heraus. In Zeile 250 werden diese beiden Variablen zunächst einmal gelöscht. Als nächstes positionieren wir den Pufferzeiger auf die Position 144 des Dir-Header-Blocks. Die 16 dort folgenden Bytes enthalten den Namen der Diskette, der mit Hilfe der Schleife in Zeile 270 in dn$ eingelesen wird. Ebenso wird mit der ID verfahren. Sie steht in den 5 Bytes ab Position 163. In Zeile 280 wird darauf positioniert und der String id$ wird in Zeile 290 eingelesen. Kommen wir nun zum zweiten Teil der Routine, in dem die Directoryeinträge eingelesen werden:

```
299 q=0
300 print#1,"u1 2 0";tr;se
305 a=int(q/8):dt(a)=tr:ds(a)=se
310 gosub600:tr=a
320 gosub600:se=a
330 forj=1to8
335 printq
340 gosub600:ty(q)=a
350 gosub600:ft(q)=a
360 gosub600:fs(q)=a
370 x$=""
380 fori=1to16:get#2,a$:x$=x$+a$:next
390 na$(q)=x$
400 fori=1to9:get#2,a$:next
410 gosub600:bl(q)=a
420 gosub600:bl(q)=bl(q)+a*256
425 get#2,a$:get#2,a$
430 q=q+1
440 next j
450 iftr<>0then300
460 :
470 close1:close2
480 q=q-1
500 ifna$(q)=""thenq=q-1:goto500
510 return
```

In Zeile 299 wird nun zunächst die Laufvariable "q" initialisiert. Sie gibt später an, wieviele Einträge in diesem Directory gefunden wurden. Der U1-Befehl in Zeile 300 liest nun den ersten Directoryblock ein, dessen Track und Sektor ja noch in TR und SE stehen. Anschließend werden beide Variablen im 0. Element von "dt" und "ds" abgelegt. Hierbei wird die Hilfsvariable "a" als Indexvariable verwendet. Ihr wird der Wert (q dividiert durch 8) zugeordnet, da in einem Block ja immer 8 Fileeinträge stehen. In den Zeilen 310 und 320 werden nun schon einmal

Track und Sektornummer des Folgeblocks in TR und SE eingelesen. Nun wird eine Schleife durchlaufen, die die acht Fileeinträge dieses Directoryblocks in den entsprechenden Feldern ablegt. Zur Information geben ich Ihnen hier noch einmal den Aufbau eines Directoryeintrags an (siehe Floppy-Kurs, Teil 4):

Byte Nummer	Aufgabe
000	Byte für den Filetyp
001 – 002	Track und Sektor des ersten Datenblocks dieses Files
003 – 018	16 Bytes für den Filenamen
019 – 020	Bei relativen Files stehen hier Track und Sektor des ersten Si-Sektor-Blocks (sonst unbenutzt)
021	Bei relativen Files seht hier die Länge eines Datensatzes (sonst unbenutzt).
022 – 025	unbenutzt
026 – 027	Zwischenspeicher des Tracks und Sektors beim Überschreiben mit dem Klammeraffen ("") vor dem Filenamen.
028 – 029	Hier steht die Länge des Files in Blocks als 16- Bit-Lo/ Hi-Zahl
030 – 031	unbenutzt

In den Zeilen 340 – 360 werden nun, obiger Liste entsprechend, der Reihe nach das Filetypenbyte, sowie Starttrack und -sektor des Files in die entsprechenden Felder eingelesen. Die anschließenden 16 Zeichen stellen den Filenamen dar und werden als Text, mit Hilfe einer Schleife in das Feld "na$" eingelesen. In Zeile 400 überlesen wir nun 9 Zeichen des Fileeintrags, die für uns nicht von Nutzen sind. In den Zeilen 410 und 420 werden nun noch Low- und High-Byte der Blockanzahl eingelesen und umgerechnet im Feld "bl" abgelegt. Hiernach werden noch die beiden letzten unbenutzten Bytes überlesen, um den Pufferzeiger auf den Anfang des nächsten Eintrags zu positionieren. Nun wird die Laufvariable q um eins erhöht und an den Schleifenanfang zurück verzweigt. Ist die Schleife 8 Mal durchlaufen worden, so haben wir alle Einträge dieses Directoryblocks eingelesen und können den nächsten Block laden. Da der letzte Directoryblock immer Track 0 als Folgetrack beinhaltet können wir mit der IF-THEN-Abfrage in Zeile 450 prüfen, ob der letzte benutzte Directoryblock schon eingelesen wurde, oder nicht. Ist "tr" ungleich null, so muß die Schleife nochmals wiederholt werden und es wird zu Zeile 300 zurückverzweigt. Im anderen Fall sind wir beim letzten Directoryblock angelangt und können die Filekanäle schließen. In Zeile 480 wird nun die Laufvariable "q" um 1 erniedrigt, da dieser Feldindex beim letzten Durchlauf ja noch keine neue Zuweisung erhielt. Zeile 500 dient nun dazu, den effektiv letzten Eintrag in diesem Directoryblock herauszusuchen, da ja nicht unbedingt alle 8 Einträge in diesem Block benutzt sein müssen. Dies geschieht einfach darin, daß wir "q" solange herunterzählen bis in der Variablen na$ (q) ein Text steht. Da Leereinträge nur Nullen enthalten, ist der Filename bei solchen Einträgen 16 Mal der Wert CHR$(0), was einem Leerstring ("") entspricht. Abschließend wird in das aufrufende Programm zurückverzweigt. In unseren Feldvariablen befinden sich nun alle Informationen des Directorys, sowie der Diskettenname in "dn$" und die ID in "id$" . In der Laufvariablen "q" steht die Anzahl der vorhandenen Einträge minus 1 (beachten Sie bitte, daß im Indexfeld 0 ebenfalls ein Eintrag enthalten ist."q" bezeichnet lediglich die Nummer des letzten Eintrags). Nun wissen wir also schon, wie wir die Directoryinformationen in den Computer bekommen.
Die Hauptroutine unseres UNDEL-Progamms muß jetzt also nur noch die Unterroutine bei Zeile 200 aufrufen um sie zu erhalten. Um jetzt alle DEL-Einträge herauszusuchen, müssen wir lediglich alle Einträge heraussuchen, die in den unteren 3 Bits des Filetypenbytes den Wert 0 stehen haben. Wir müssen uns deshalb auf die unteren 3 Bits beschränken, da in den Bits 7 und 6 ja auch noch Informationen über das File gespeichert sind. In den Zeilen 1100 – 1120 des

Hauptprogramms werden diese Einträge nun herausgesucht und deren Feldindizes im Feld "li" abgelegt. Das Hauptprogramm von UNDEL beginnt übrigens bei Zeile 1000 . Aus Platzgründen will ich Ihnen ab jetzt nur noch die Zeilen auflisten, die für uns interessant sind. Der Rest dient hauptsächlich der Bildschirmformatierung und soll uns hier deshalb nicht interessieren. Das ganze Programm können Sie sich ja einmal auf dieser MD anschauen.

Hier nun die Zeilen 1100 – 1120:

```
1100  z=0 1105 fori=0 toq
1110  if( ty( i) and7)=0 thenli( z)= i: z= z+1
1120 next
```

Wie Sie sehen, benutzen wir hier die Laufvariable "z" als Indexvariable für das Feld "li" . Mit dem Ausdruck "ty (i) and 7" isolieren wir die unteren drei Bits des Typenbytes und löschen die evtl. gesetzten Bits 7 und 6 für den IF-THEN-Vergleich. Wenn "z" nun größer Null ist, so wurden DEL-Files gefunden. Selbige werden in den Zeilen 1190 – 1230 auf dem Bildschirm ausgegeben und es wird gefragt, welches davon gerettet werden soll. In den Zeilen 1240 bis 1350 wird nun nach dem Typ des Files gefragt und ob es evtl. auch vor erneutem Löschen geschützt werden soll. Hier wieder ein Programmauszug:

```
1240  gosub100:print"Zu retten:";na$(li)
1250  print"0 - DEL"
1260  print"1 - SEQ"
1270  print"2 - PRG"
1280  print"3 - USR"
1290  print"4 - REL"
1300  input"Welchen Filetyp soll ich zuordnen";t1
1310  if(n<0)or(n>4)then1300
1320  print"File auch schuetzen (J/N) ?"
1330  geta$:ifa$=""then1330
1340  ifa$="j"thent1=t1+64
1350  t1=t1+128:ty(li(n))=t1
```

Wie Sie sehen, wird in die Variable "t1" eine Zahl zwischen 0 und 4 eingelesen. Die Zahlen, die die einzelnen Typen zuordnen, entsprechen den Codes, die auch die Floppy zur Erkennung des Filetyps benutzt. In den Zeilen 1320 –1340 wird nun gefragt, ob das File auch vor dem Löschen geschützt werden soll. Dies regelt Bit 6 im Filetyp-Byte. Soll das File geschützt werden, so wird zu der Variablen "t1" der Wert 64 hinzuaddiert, was dem Setzen des 6. Bits entspricht. Zusätzlich müssen wir noch Bit 7 setzen, das anzeigt, daß das File ein gültiges File ist. Dies geschieht durch addieren des Wertes 128 . Nun müssen wir nur noch den neuen Wert für den Filetyp in die Feldvariable "ty" übernehmen. Als nächstes muß der neue Filetyp im Directory eingetragen und die Datenblocks des Files als 'belegt' gekennzeichnet werden. Dies geschieht in den Zeilen 1370 – 1550. Hier wieder ein Auszug:

```
1370  gosub100:print"Schreibe neuen Eintrag"
1380  bl=int(li(n)/8):ei=li(n)-bl*8
1390  tr=dt(bl):se=ds(bl)
1400   open1,8,15,"i":open2,8,2,"#"
1410  print#1,"u1 2 0";tr;se
1420  po=2+ei*32
1430  print#1,"b-p 2";po
1440  print#2,chr$(t1);
1450  print#1,"u2 2 0";tr;se
1460 :
```

In den Zeilen 1380 – 1390 wird nun zunächst den Variablen TR und SE die Werte für den Directoryblock, in dem der Eintrag steht zugewiesen. Als nächstes öffnen wir einen Befehlsund einen Pufferkanal mit den logischen Filenummern 1 und 2 (wie auch schon in der Dir-Lese-

Routine). In Zeile 1410 wird der Block des Eintrags eingelesen. Nun muß der Pufferzeiger auf den Anfang des Eintrags positioniert werden. Da jeder Fileeintrag 32 Bytes lang ist, errechnet sich seine Anfangsposition aus seiner Position im Block (steht in der Variablen "ei", die in Zeile 1380 definiert wurde) multipliziert mit 32. Zusätzlich müssen wir noch zwei Bytes aufaddieren, in denen Trackund Sektornummer des Folgeblocks stehen (ganz am Anfang des Directoryblocks). In Zeile 1430 wird nun auf die errechnete Position positioniert. Nun müssen wir an dieser Stelle nur noch das neue Typenbyte eintragen (geschieht in Zeile 1440) und den Directoryblock mittels "U2" wieder auf die Diskette zurückschreiben. Schon ist der Fileeintrag geändert!
Zum Schluß müssen wir noch alle vom File belegten Blocks heraussuchen und wieder neu belegen. Den ersten davon finden wir in den Feldern "ft" und "fs". Hier wieder ein Programmauszug:

```
1465  z=0
1470  print"Belege Blocks neu... "
1480  tr=ft(li(n)):se=fs(li(n))
1490  print#1,"b-a 0";tr;se
1500  print#1,"u1 2 0";tr;se
1505  z=z+1
1510  gosub600:tr=a
1520  gosub600:se=a
1530   iftr<>0then1490
1540  close1:close2
1550  printz;"Blocks gefunden."
```

Hier wird die Variable "z" dazu benutzt, die gefundenen Blocks zu zählen. In Zeile 1480 werden nun die Feldeinträge in "ft" und "fs" in TR und SE Übertragen. Zeile 1490 benutzt nun den Block-Allocate-Befehl um diesen Block als 'belegt' zu kennzeichnen. In Zeile 1500 wird er dann in den Puffer gelesen, da seine ersten beiden Bytes ja auf den nächsten Block des Files zeigen. Dessen Track und Sektornummer wird nun in den Zeilen 1510 und 1520 in TR und SE eingelesen und für den nächsten Durchlauf der Schleife verwendet. Sollte TR jedoch den Wert 0 aufweisen, so sind wir beim letzten Block angelangt und können die Blockbelegung beenden. Dies regelt die IF-THEN-Anweisung in Zeile 1530. Somit wären alle wichtigen Teile des UNDEL-Programms besprochen. Um einen Gesamtüberblick zu erhalten, rate ich Ihnen, sich das Programm auf dieser MD einmal auszudrucken und anzuschauen.
Übrigens beinhaltet das Programm noch einen kleinen Schönheitsfehler: Wenn Sie nämlich z.B. 9 Files auf einer Diskette haben, und das letzte davon löschen, so findet es die Directory-Lese-Routine nicht mehr. Das liegt daran, daß nach dem Löschen des Eintrags der zweite Directoryblock ja nicht mehr benötigt wird und deshalb als Folgeblock Track 0, Sektor 255 im ersten Directoryblock eingetragen wird. Um diesen Fehler zu beheben, können Sie ja eine Routine schreiben, die alle Directoryblocks der Reihe nach einliest. Das ist zwar sehr zeitaufwendig, könnte jedoch den Fehler beheben. Sie können dabei davon ausgehen, daß die Folgetracks und -sektoren beim Directory immer gleich sind. Bei einem vollen Directory (144 Einträge), können Sie so diese Einträge mit Hilfe eines Diskettenmonitors einfach herausfinden.
Ich verabschiede mich hiermit von Ihnen bis zur nächsten Ausgabe, wo wir uns an die Floppyprogrammierung in Assembler heranwagen möchten.

(ub)

Teil 7 – Magic Disk 12/92

Hallo und herzlich Willkommen zum siebten Teil des Floppy-Kurses. In diesem Monat möchten wir uns an die Floppy-Programmierung in Assembler heranwagen. Hierbei werden wir alle notwendigen Routinen zum I/ O-Handling von Maschinenspache aus kennenlernen. Allerdings sollten Sie schon einige Assembler-Vorerfahrung mitbringen und zumindest Kenntnis vom

Befehlssatz des 6510-Prozessors haben.

DER EINSTIEG IN ASSEMBLER

Wie Sie bemerkt haben, waren alle bisherigen Programmbeispiele und Befehlserläuterungen in BASIC programmiert. Dies diente hauptsächlich der Einfachheit halber. Wenn Sie nun ein angehender Assemblerprogrammierer sind, so werden Sie sich desöfteren schon einmal gefragt haben, wie Sie die Floppy in Maschinensprache ansprechen können. Dies unterscheidet sich von BASIC zwar schon in einigen Punkten, da es etwas aufwendiger ist, jedoch können wir hier komplett auf schon im Betriebssystem des 64ers vorhandene Routinen zurückgreifen, die lediglich mit den richtigen Parametern gefüttert werden müssen. Diese Routinen sind dieselben, die auch vom BASIC des C64 benutzt werden, weshalb die Parameterübergabe der von BASIC sehr ähnlich ist. Kommen wir zunächst einmal zu einigen grundlegenden Dingen:

ÖFFNEN UND SCHLIESSEN VON FILES

Jedesmal, wenn wir einen Zugriff auf die Floppy ausführen wollten, mussten wir zuvor immer einen Filekanal öffnen. Hierzu wurden mit Hilfe des OPEN-Befehls einige Parameter an die Floppy übergeben, die zur eindeutigen Definition des Kanals notwendig waren. Diese Parameter waren die Filenummer, die Devicenummer und die Sekundäradresse. Oft folgte der ganzen Anweisung dann auch noch ein Filename, dessen Appendix ebenso die verlangte Operation spezifizierte. Man kann also sagen, daß diese Werte alles Grundwerte sind, die zum Datenaustausch mit der Floppy benötigt werden. Aus diesem Grund müssen sie auch bei jeder Fileoperation angegeben werden. Möchten wir nun eine Operation von Assembler aus durchführen, so müssen wir diese Parameter zunächst dem Betriebssystem mitteilen. Dies geschieht über die beiden ROM-Routinen "SETPAR" und "SETNAM". SETPAR hat die Einsprungadresse $FFBA. An sie werden die Filenummer, sowie Geräte und Sekundäradresse übergeben. SETNAM legt den Filenamen (inklusive Appendix) fest. Sie steht bei $FFBD. Beide Routinen tun nichts anderes, als die gegebenen Werte in Betriebssystem-Zwischenspeichern der Zeropage abzulegen.

Nachdem die Werte festgelegt sind, können wir die Betriebssystemroutine "OPEN" aufrufen, die uns den Kanal mit der Floppy öffnet. Ihre Einsprungadresse ist bei $FFC0. Zum Schließen benutzen wir die Routine "CLOSE" bei $FFC3.

Wenn wir ein File öffnen, so müssen wir zunächst einmal seinen Namen festlegen. Diesen legen wir irgendwo im Speicher, an einer bekannten Adresse, ab. Die Länge des Namens benötigen wir ebenso. Als nächstes können wir SETNAM und SETPAR aufrufen. SETNAM wird die Adresse des Filenamens und seine Länge, SETPAR die logische Filenummer, die Geräte- und die Sekundäradesse übergeben. Hier eine Übersicht der Registerbelegungen:

SETPAR	Akku	Filenummer
	X-Register	Geräteadresse
	Y-Register	Sekundäradresse
SETNAM	Akku	Länge des Filenamens
	X-Register	Low-Byte der Speicheradresse des Namens
	Y-Register	High-Byte der Speicheradresse des Namens

Ich möchte Ihnen hierzu ein Beispiel liefern. Angenommen, Sie wollten das sequentielle File "TEST" zum Lesen öffnen. Als Filenummer wollen wir die 1 wählen, als Sekundäradresse 2. Die Geräteadresse ist in Bezug auf die Floppy natürlich 8 (oder 9,10,11 wenn Sie eine zweite, dritte oder vierte Floppy besitzen). All diese Parameter entsprechen also dem BASIC-Befehl:

```
OPEN 1,8,2,"TEST,S,R"
```

Möchten wir diesen Befehl nun in Maschinensprache umsetzen, so schreiben wir folgendes

Assembler-Programm. Hierbei gehe ich davon aus, daß wir den Filenamen "TEST,S,R" als ASCII-Code bei Adresse $0334 (dem Kasettenpuffer) abgelegt haben:

```
LDA #$01        ;Filenummer 1
LDX #$08        ;Geräteadresse 8
LDY #$02        ;Sekundäradresse 2
JSR $FFBA       ;SETPAR aufrufen
LDA #$08        ;Filename ist 8 Zeichen lang
LDX #$34        ;und liegt bei Adresse
LDY #$03        ;$0334
JSR $FFBD       ;SETNAM aufrufen
```

Die Parameter für das File "TEST" wären damit festgelegt. Da das ",S,R" im Filenamen enthalten ist, weiß die Floppy auch gleich schon, daß wir ein sequentielles File lesen möchten. Als Nächstes müssen wir das zuvor spezifizierte File öffnen. Dies geschieht über die oben schon erwähnte Betriebssystemroutine "OPEN". Sie benötigt keine Parameter und wird direkt aufgerufen:

```
JSR $FFC0       ;File mit zuvor gesetztem Namen und Parametern öffnen
```

Nun ist das File "TEST" geöffnet. Da aber auch jeder Datenfluß einmal ein Ende hat, muß unser File irgendwann einmal wieder geschlossen werden. Dies geschieht, wer hätte das gedacht, mit der ROM-Routine "CLOSE", die bei Adresse $FFC3 angesprungen wird. Da auch mehrere Files gleichzeitig offen sein können, muß ihr die Filenummer des zu schließenden Files im Akku übergeben werden:

```
LDA #$01        ;File mit Filenummer 1
JSR $FFC3       ;schließen
```

DER DATENAUSTAUSCH

Wie man ein File öffnet, sollte Ihnen nun klar sein. Wenn Sie jedoch auch noch mit ihm kommunizieren wollen (was sie bestimmt tun möchten, da es anders sinnlos wäre ein File zu öffnen), so müssen Sie weiterhin fünf Betriebssystem-Routinen kennen, die Ihnen dies ermöglichen. Zunächst wären da "CHKIN" und "CHKOUT". Sie dienen der Umleitung der Standard Ein- / Ausgabe auf den entsprechenden Filekanal. Wenn Sie aus einem File lesen wollen, so sollten Sie nach dem Öffnen desselben die Routine CHKIN aufrufen. Mit ihr teilen Sie dem Betriebssystem mit, daß Sie, wenn Sie jetzt etwas einlesen, die Daten aus diesem Filekanal haben möchten. Möchten Sie in ein File schreiben, so müssen Sie CHKOUT aufrufen um in das geöffnete File schreiben zu können. Beide Routinen benötigen die Filenummer des entsprechenden Files im X-Register. Die Einsprungadresse von CHKIN ist $FFC6, die von CHKOUT $FFC9.

Wenn Sie die Standard-Ein / Ausgabe mit Hilfe unserer beiden Routinen umgeleitet haben, so können Sie die Ihnen vielleicht schon bekannten Routinen BASIN ($FFCF) und BASOUT ($FFD2) zum Lesen und Schreiben von Daten vom, bzw. an, das Ein- / Ausgabe-Gerät benutzen. Rufen Sie diese Routinen bei unveränderter Standard Ein- / Ausgabe auf, so erscheint bei BASIN ein Cursor auf dem Bildschirm, der dem Assemblerprogramm eine Eingabe von der Tastatur übermittelt, bei BASOUT wird ein Zeichen an die aktuelle Cursorposition gedruckt und der Cursor um eine Stelle weiterbewegt (wie Sie bemerken ist das Standard-Eingabegerät die Tastatur, das Standard-Ausgabegerät der Bildschirm). Bei umgeleiteter Eingabe erhalten Sie nun beim Aufruf von BASIN das aktuelle Byte des geöffneten Files im Akku zurück. Bei umgeleiteter Ausgabe wird jedes Byte, daß Sie in den Akku laden und anschließend an BASOUT übergeben in das geöffnete File hineingeschrieben. Haben Sie nun Ihre Fileoperationen beendet, so ist es notwendig, die fünfte Routine zu benutzen, von der ich oben sprach. Sie heißt "CLRCH" und wird verwendet, um die Standard Ein- / Ausgabe wieder zurückzusetzen auf 'Tastatur' und 'Bildschirm'. Ihre Einsprungadresse ist $FFCC. Sie wird VOR

dem Aufruf von CLOSE benutzt und benötigt keine Parameter.

FEHLERERKENNUNG UND BEHANDLUNG

Bevor wir uns der Praxis zuwenden, zunächst noch ein Wort zur Fehlererkennung. Hierüber können wir nämlich beim Lesen eines Files auch erkennen, wann wir sein letztes Byte gelesen haben.

Prinzipiell gilt: tritt während der Arbeit einer der Floppy-Betriebssystem- Routinen ein Fehler auf, so wird das an das aufrufende Programm durch ein gesetztes Carry-Bit zurückgemeldet. So können Sie also nach jeder der Routinen durch eine einfache "BCS"- Verzweigung ("Branch on Carry Set") auf eine Fehlerbehandlungsroutine verzweigen. Hierbei steht dann im Akku ein Fehlercode, mit dessen Hilfe Sie die Art des Fehlers feststellen können. Hier eine Liste mit den möglichen Fehlermeldungen und den Ursachen für einen aufgetretenen Fehler. In eckigen Klammern stehen jeweils die Betriebssystem-Routinen, die den entsprechenden Fehler auslösen können. Steht nichts dahinter, so handelt es sich um einen allgemeinen Fehler:

0	"Break Error"	Die RUN/ STOP-Taste wurde gedrückt.
1	"too many files"	Der 64er verwaltet intern maximal 10 offene Files. Sie versuchten ein elftes File zu öffnen. Oder aber sie haben schon zu viele Files zur Floppy hin offen (Sie erinnern sich: man darf maximal 3 sequentielle, oder 1 relatives und 1 sequentielles File gleichzeitig offen halten). [OPEN]
2	"file open"	Sie versuchten, ein schon offenes File nochmal zu öffnen.
3	"file not open"	Sie versuchten, ein ungeöffnetes File anzusprechen. [CHKIN, CHKOUT, CLOSE]
4	"file not found"	Das File, das Sie zum Lesen öffnen wollten existiert gar nicht. [OPEN]
5	"device not present"	Die Floppy (oder das gewählte Gerät) ist nicht eingeschaltet. [OPEN]
6	"not an input file"-	Sie versuchten aus einem zum Schreiben geöffneten File zu lesen. [CHKIN]
7	"not an output file"	Sie versuchten in ein zum Lesen geöffneten File zu schreiben. [CHKOUT]
8	"missing filename"	Sie gaben keinen Filenamen an. [OPEN]
9	"illegal device number"	Sie gaben eine ungültige Devicenummer an. [OPEN]

Beachten Sie bitte, daß nicht unbedingt alle Fehler aus obiger Liste auftreten müssen, da man die oben genannten Routinen auch zum Ansprechen von anderen Geräten verwenden kann (z. B. Drucker, Datasette, etc.). Öffnen Sie z.B. einen Kanal zum Drucker, so brauchen Sie keinen Filenamen - der Fehler 8 wird in dem Fall also nie auftreten. Als weitere Fehlererkennungshilfe stellt uns das Betriebssystem auch eine Statusspeicherstelle zur Verfügung. Sie ist absolut identisch mit der Variablen "ST" von BASIC. Fragt ein BASIC-Programm diese Variable ab, so greift der BASIC-Interpreter auf eben diese Speicherstelle zurück. Die Assemblerprogrammierer finden den I/ O-Status in der Zeropage, nämlich in Speicherstelle $90 (dez. 144). Um feststellen zu können, ob während der Arbeit mit der Floppy ein Fehler aufgetreten ist, müssen wir sie lediglich einlesen und analysieren. Ein Fehler wird hierbei durch das gesetzt sein eines oder mehrerer Bits dieser Speicherstelle gemeldet. Hier eine Belegung der Bits (nur für die Arbeit mit der Floppy gültig, bei Kasettenbetrieb gilt eine andere Belegung):

Bit	**Bedeutung**
0	Fehler beim Schreiben
1	Fehler beim Lesen
6	Datei-Ende wurde erreicht (Lesen)
7	Gerät nicht vorhanden oder abgeschaltet ("device not present")

Sie sehen, daß Sie hier, wie auch in BASIC, das Ende eines Files durch gesetzt sein des 6. Bits von ST feststellen können. Dann nämlich hat ST den Wert 64, den wir bei BASIC-Abfragen auch immer verwendeten. Sie sehen übrigens auch, warum bei erneutem Lesen trotz beendetem File die Fehlernummer 66 zurückgeliefert wird. Das File ist dann nämlich zu Ende und es trat ein Fehler beim Lesen auf, weil es ja gar nichts mehr zu lesen gibt. Damit sind die Bits 1 und 6 gesetzt (2+64) was dem Wert 66 entspricht. Wenn ST den Wert 0 enthält, so ist alles gut gegangen. Auf diese Weise können wir in Assembler sehr einfach den Fehler abfragen: Wir lesen die Speicherstelle $90 einfach ein und verzweigen mittels BNE auf eine Fehlerbehandlungsroutine.

ZUSAMMENFASSUNG

Abschließend zu diesen Routinen möchte ich Ihnen nun zwei Universal-Lese/ Schreib-Routinen vorstellen, die Sie zum Lesen, bzw. Schreiben eines Files verwenden können. Sie finden diese Routinen auch auf dieser MD unter dem Namen "FK.I/O.S". Sie sind im Hypra-Ass- Format gespeichert. Um sie sich anzuschauen können Sie sie wie ein BASIC-Programm laden und listen.

Zunächst einmal möchte ich Ihnen die "ReadFile"- Routine vorstellen. Sie liest ein File an eine beliebige Adresse ein. Hierbei übergeben Sie Low- und High-Byte in X und Y-Register, sowie die Länge des Filenamens im Akku. Der Filename selbst soll immer bei $0334 (dez.820) stehen:

ReadFile:

```
                stx  $fb            ;Startadresse in
                sty  $fc            ; Zeiger schreiben

                ldx  #$34           ;Namensadresse=$0334
                ldy  #$03           ;in X/Y (Len in Ak.)
                jsr  $ffbd          Und Name setzen

                lda  #01            ;Filenummer=1
                ldx  #08            ;Geräteadresse=1
                ldy  #00            ;Sek.=0 (=PRG lesen)
                jsr  $ffba          ;Parameter setzen

                jsr  open           ;File öffnen
                ldx  #01            ;Ausgabe auf FNr. 1
                jsr  chkin          ; umleiten

                ldy  #00            ;Index auf 0 setzen
loop1:          jsr  basin          ;Byte lesen
                sta  ($fb),y        ;und auf Zeigeradresse speichern

                inc  $fb            ;Den Zeiger in
                bne  l1:            ;$FB/$FC um 1
                inc  $fc            ;erhöhen

l1:             lda  $90            ;File-Ende erreicht?
                beq  loop1          ;Nein, dann weiter!

                jsr  $ffcc          ;Ja, also Standard Ein-/Ausgabe zurücksetzen.

                lda  #01            ;Und File mit File-
                jmp  $ffc3          ;Nummer 1 schließen
```

Als nächstes stelle ich Ihnen die "WriteFile"-Routine vor. Sie speichert Daten in einem beliebigen Speicherbereich auf Diskette und wird mit denselben Voraussetzungen aufgerufen wie "ReadFile": Name bei $0334, Namenslänge im Akku und Startadresse des zu speichernden Bereichs in X- / Y-Register. Zusätzlich müssen Sie die Endadresse dieses Bereichs (in Lo- / Hi-Darstellung) vorher in den Speicherstellen $FD / $FE abgelegt haben:

WriteFile:

```
                stx  $fb            ;Startadresse in
                sty  $fc            ;Zeiger schreiben
                ldx  #$34           ;Namensadresse=$0334
                ldy  #$03           ;in X/Y (Len in Ak.)
                jsr  $ffbd          ;Und Name setzen

                lda  #01            ;Filenummer=1
                ldx  #08            ;Geräteadresse=1
                ldy  #01            ;Sek.=1 (=PRG schreiben)

                jsr  $ffba          ;Parameter setzen

                jsr  $ffc0          ;File öffnen
                ldx  #01            ;Eingabe auf Fnr.1
                jsr  $ ffc9         ;umleiten

                ldy  #00            ;Index auf 0 setzen
loop2:          lda  ($fb),y        ;Byte aus Sp. Holen
                jsr  $ffd2          ;und an Floppy senden

                inc  $fb            ;Den Zeiger in
                bne  l2             ;$FB/$FC um 1
                inc  $fc            ;erhöhen
l2:             lda  $fe            ;Quellzeiger $FB/$FC
                cmp  $fc            ;mit Zielzeiger
                bne  loop2          ;$FD/$FE

                lda  $fd            ;vergleichen. Wenn
                cmp  $fb            ;ungleich, dann
                bne  loop2          ;weitermachen.

                jsr  $ffcc          ;Wenn gleich, dann Standard-I/O zurücksetzen.

                lda  #01            ;und File mit File-
                jmp  $ffc3          ;nummer 1 schließen
```

Wie Sie sehen, habe ich hier den Trick mit den Sekundäradressen verwendet. Da bei der Sekundäradresse die Werte 0 und 1 für "Programm lesen" bzw."Programm schreiben" stehen, erübrigt sich ein umständliches anhängen von ",P,R" bzw. ",P,W" an den Filenamen. Dafür jedoch können mit den Routinen nur PRG-Files gelesen und geschrieben werden.

Sie können diese Routinen nun universell verwenden, um Programmdaten (z.B. die High-Score-Tabelle in einem Spiel) auf Diskette zu schreiben und wieder von ihr zu lesen. Aber Achtung: die Routinen sind nur zum reinen Lesen und Schreiben gedacht! Files, die normalerweise mit LOAD in den Rechner geladen werden, oder mit SAVE abgespeichert wurden, können nicht ganz so behandelt werden! Hierbei müssen Sie beachten, daß die SAVE-Routine des Betriebssystems immer die Anfangsadresse des Files in den ersten beiden Bytes (in Lo-/ Hi-Byte-Darstellung) mitspeichert. Lesen Sie ein solches File mit "BASIN" ein, so erhalten Sie in den ersten beiden Bytes immer zuerst diese Adresse. Beachten Sie also, daß diese beiden Bytes nicht zu den eigentlichen Daten des Files gehören! Ebenso können Sie kein mit "WriteFile" geschriebenes File mit dem LOAD-Befehl laden, weil hierbei nämlich die ersten beiden Bytes als Startadresse gewertet werden und das File anschließend irgendwohin in den Speicher geladen wird! Zum Kopieren von Files sind diese Routinen wiederum optimal geeignet, da sie die Ladeadresse ja mitladen und mitspeichern.
Sie können sie aber auch so modifizieren, daß diese Adresse mitberücksichtigt wird. Hierzu müssen Sie "ReadFile" lediglich so umprogrammieren, daß sie die ersten beiden Bytes liest und als Anfangsadresse nimmt, bzw. daß "WriteFile" die gegebene Anfangsadresse, die in X- und Y-

Register übergeben wird, vorher an das zu speichernde File sendet. Noch einfacher geht das aber mit den Betriebssystemroutinen für LOAD und SAVE. Sie nehmen uns diese Arbeit nämlich komplett ab und sind zudem noch relativ flexibel zu handhaben.

LOAD UND SAVE

Wie schon erwähnt stellt uns das Betriebssystem ebenso zwei Routinen zur Verfügung, mit denen wir komplette Files nachladen, bzw. speichern können. Der Aufruf ist hierbei sehr einfach. Zunächst einmal müssen wir unsere Files wieder mittels SETNAM und SETPAR spezifizieren. Danach werden die Prozessorregister einfach nur noch mit den entsprechenden Parametern gefüttert und die benötigte Routine wird aufgerufen.
Kommen wir zuerst zur LOAD-Routine. Sie liegt bei $FFD5 und hat folgende Aufrufparameter:

Akku:	Operationsflag ($00 oder $01)
X-Register:	Low-Byte der Ladeadresse
Y-Register:	High-Byte der Ladeadresse

Das Operationsflag, das in den Akku kommt, hat folgende Bewandtnis: die LOAD-Routine kann nämlich auch zum Verifizieren eines Files verwendet werden (BASIC- Befehl "VERIFY"). Hierbei wird haargenau so verfahren wie beim Laden, jedoch mit dem Unterschied, daß das File nicht in den Speicher geschrieben, sondern nur gelesen wird. Möchte man nun laden, so muß der Wert 0 im Akku stehen. Möchte man verifizieren, so gehört der Wert 1 in ihn hinein. Desweiteren kann in X- und Y-Register die Startadresse des File übergeben werden, an die es geladen werden soll. Die LOAD-Routine überliest dann ganz einfach die ersten beiden Bytes, die diese Adresse ja angeben, und liest das File anschließend an die gegebene Adresse. Möchten Sie das File jedoch an seine vorgegebene Adresse laden, so müssen Sie X- und Y-Register lediglich mit 0 füllen.
Zum besseren Verständnis einmal eine kleine Routine, die ein File mittels LOAD-Routine an seine im File vorgegebene Adresse lädt. Sie verlangt als Aufrufparameter die Filenamenlänge im Akku und den Filenamen bei Adresse $0334:

```
LDX  #$34          ;Name bei $0334 (Länge ist
LDY  #$03          ;noch vom Aufruf im Akku)
JSR  $FFBD         ;Namen setzen (SETNAM)
LDA  #01           ;Filenummer=1
LDX  #08           ;Gerät Nr. 8
LDY  #00           ;Sek.=0 für "PRG laden"
JSR  $FFBA         ;Parameter setzen (SETPAR)
LDA  #00           ;Load-Flag in Akku
TAX                ;X/Y löschen (Startadresse
TAY                ;aus dem File holen)
JMP  $FFD5         ;und laden
```

Wie Sie sehen, werden SETNAM und SETPAR genauso benutzt wie in der "ReadFile"-Routine. Im Prinzip können Sie die Angabe einer Filenummer weglassen, da Sie sie selbst ja nicht brauchen. Es wird in dem Fall gerade die Nummer gewählt, die im Akku steht. Nur hat das den Nachteil, daß der Ladevorgang nicht immer funktioniert, wenn Sie noch weitere Files offen halten. Das File steht am Ende der Routine an der Stelle im Speicher, die von seinen ersten beiden Bytes spezifiziert wurde.
Möchten Sie es allerdings an eine bestimmte Adresse laden, z.B.$1234, so müssen Sie die beiden Befehle "TAX" und "TAY" durch die folgenden beiden Zeilen ersetzen:

```
LDX #$34
LDY #$12
```

Kommen wir nun zur SAVE-Routine des Betriebssystems. Sie wird ähnlich aufgerufen. Zunächst müssen wieder mittels SETNAM und SETPAR der Name und die Parameter des Files

gesetzt werden. Anschließend kann die SAVE-Rotine aufgerufen werden. Sie benötigt jedoch ein paar mehr Parameter, nämlich Start- und Endadresse. Da hierfür die drei Prozessorregister nicht ausreichen, wurde folgender Weg gegangen: Zunächst legen Sie die Startadresse des zu speichernden Bereichs in Lo-/ Hi-Darstellung in zwei aufeinanderfolgenden Adressen der Zeropage ab.
Hierzu bieten sich $FB und $FC an, da sie vom Betriebssystem nicht benutzt werden. Nun laden Sie einen "Zeropage-Zeiger" in den Akku. Dieser ist nichts anderes als die Nummer der Speicherstelle in der Sie das Low-Bytes der Startadresse abgelegt haben. In unserem Beispiel also $FB. Die Endadresse des zu speichernden Bereichs legen Sie in X- und Y-Register ab und rufen anschließend die SAVE-Routine auf. Ihre Einsprungadresse liegt bei $ FFD8. Hier wieder eine Beispielroutine. Sie verlangt wieder den Filenamen bei $0334 und dessen Länge im Akku. Desweiteren müssen Sie die Startadresse in $FB/$FC abgelegt haben und die Endadresse in X- und Y-Register übergeben:

```
STX  $FD          ;Endadresse in $FD/$FE
STY  $FE          ;sichern
LDX  #$34         ;Filenamen bei $0334
LDY  #$03         ;setzen
JSR  $FFBD        ;(SETNAM)
LDA  #01          ;Filenr. 1
LDX  #08          ;Geräte Nr. 8
LDY  #01          ;Sek.=1 für "PRG speichern"
JSR  $FFBA        ;Parameter setzen (SETPAR)
LDA  #$FB         ;Zeiger auf Startadr. laden
LDX  $FD          ;Endadresse wieder in
LDY  $FE          ;X- und Y-Reg. holen
JMP  $FFD8        ;Und speichern!
```

Wie Sie sehen, wird die Endadresse hierbei in $FD/$FE zwischengespeichert, um den Aufruf korrekt durchführen zu können. Diese beiden Speicherstellen sind übrigens ebenfalls vom Betriebssystem unbenutzt.
Die beiden Programmbeispiele zum Laden und Speichern mittels LOAD und SAVE finden Sie übrigens ebenfalls in dem File "FK.I/O.S" auf dieser MD. Ich verabschiede mich nun wieder einmal von Ihnen, und wünsche Frohe Weihnachten und ein schönes neues Jahr, in dem wir dann tiefer in die Assemblerprogrammierung der Floppy einsteigen und einige Programmbeispiele des täglichen Gebrauchs kennenlernen werden.

(ub)

Teil 8 – Magic Disk 01/93

Herzlich Willkommen zum achten Teil des Floppy-Kurses. In diesem Monat wollen wir noch etwas weiter in die Floppyprogrammierung in Assembler einsteigen. Im letzten Monat hatten wir ja schon angesprochen, wie man ein File öffnet und mit ihm kommuniziert, sowie man Files lädt und speichert. Diesmal will ich Ihnen eine andere, effektiviere Methode zeigen, mit der man die Floppy ansprechen kann.

DIE ALTE METHODE
In der letzten Ausgabe des Floppy-Kurses hatten wir gelernt, wie man ein File öffnet, es zur Datenübertragung bereitmacht, Daten an es sendet und von ihm empfängt, und es anschließend wieder schließt. Die dort behandelte Arbeitsweise war die einfachste, die man zum reinen Lesen oder Schreiben eines Files anwendet. Es gibt jedoch eine weitere Methode Files anzusprechen. Diese ist zwar ein wenig umständlicher, dafür aber schneller und weitaus flexibler.

Rekapitulieren wir: Nach der ersten Methode gingen wir immer folgendermaßen vor:

1. "SETNAM" mit den Parametern für den Filenamen aufrufen.
2. "SETPAR" mit den Datenkanalparametern aufrufen.
3. Mit "OPEN" das File öffnen

4a. War das File eine Ausgabedatei, so mussten wir "CKOUT" aufrufen.
4b. War das File eine Eingabedatei, so mussten wir "CHKIN" aufrufen.
5a. War das File eine Ausgabedatei, so wurden die Daten mit Hilfe von "BSOUT" geschrieben.
5b. War das File eine Eingabedatei, so wurden die Daten mit Hilfe von "BASIN" gelesen.

6. Abschließend wurde mittels "CLRCH" die Standardausgabe wieder zurückgestellt.
7. Das File wurde mit "CLOSE" geschlossen.

Wie gesagt, ist diese Methode die einfachste, wenn Sie ein einziges File lediglich Lesen oder Schreiben wollen. Wenn Sie nun aber mehrere Files offenhalten und ansprechen wollen, oder aber auf einem Kanal Lesen und Schreiben wollen (so wie das beim Befehlskanal unter Mitbenutzung von Pufferkanälen oft der Fall ist), so versagt diese Methode. Das liegt daran, daß Sie mit der einfachen Methode eigentlich nicht wirklich mit den Dateien kommunizieren. Dies geschieht dann nämlich mit einem Umweg über das Betriebssystem. Das Aufrufen von "CKOUT" entspricht zum Beispiel dem Basic-Befehl "CMD". Sie leiten damit die Daten, die normalerweise auf dem Bildschirm ausgegeben werden, auf den Floppykanal um. Nun ist es aber wiederum nicht so einfach, einen zum Ausgabekanal erklärten Filekanal, als Eingabekanal zu benutzen. Ein folgendes " CHKIN" zeigt nämlich keine Wirkung. Auch wenn Sie zwei Files offenhalten, so funktioniert das Umschalten zwischen diesen beiden Files nicht immer. Der Grund dafür liegt in der Art und Weise wie die Standard-Ein/ Ausgabefiles im C64 verwaltet werden.

DIE NEUE METHODE

Um nun effizienter mit der Floppy zu kommunizieren, müssen wir auf spezielle Betriebssystemroutinen zurückgreifen, die eigens für die Datenkommunikation mit der Floppy (bzw. dem IEC-Bus, an dem Floppy und Drucker angeschlossen sind), dienen. Diese Routinen werden im Prinzip auch bei Verwendung der ersten Methode vom Betriebssystem benutzt, jedoch muß ein Standard-Kanal eben auch andere, virtuelle, Geräte (wie Bildschirm, Tastatur, etc.) ansprechen können, weshalb die Restriktionen dort etwas umständlicher sind. Deshalb ist eine Ausgabe über BASOUT, oder die Eingabe über BASIN, auch immer langsamer, als wenn Sie die entsprechenden I/ O-Routinen für den IEC-Bus benutzen, da bei den beiden ersten Routinen erst einmal entschieden werden muß, welche effektiven Ein-/ Ausgaberoutinen nun wirklich benutzt werden müssen (in unserem Fall nämlich die IEC-Busroutinen) .
Kommen wir nun jedoch endlich zu den neuen Routinen und deren Funktionsprinzip. Hierzu müssen wir uns zunächst noch einmal verdeutlichen, daß die Floppy im Prinzip ein eigenständiger Computer ist, der nichts anderes tut, als darauf zu warten, daß Befehle über den IEC-Bus kommen, um sie auszuführen. Hierbei meine ich nicht die Floppy-Befehle, wie wir sie bisher kennengelernt haben. Diese stellen wiederum eine höhere Ebene von Befehlen dar. Da der IEC-Bus als Kommunikationsschnittstelle zwischen C64 und Floppy dient, werden auf Ihm Daten entweder vom Rechner zur Floppy oder umgekehrt hin- und herbewegt. Die Verarbeitung dieser Daten ist dann dem jeweiligen Gegengerät überlassen, daß die Daten empfangen hat. Noch dazu kann es vorkommen, daß sich bis zu 7 Geräte (der C64, max.2 Drucker und max. vier Floppys) über ein und denselben Bus miteinander unterhalten müssen. In dem Fall muß vor einer Datenübertragung natürlich festgelegt werden welches der Geräte nun mit welchem (im Normalfall dem 64er) kommunizieren soll. Hierzu dienen nun insgesamt sechs Betriebssystemroutinen.
Als erstes müssen wir wieder einmal ein File auf die gewohnte Art und Weise öffnen. Zunächst übergeben wir die Filenamenparameter mit Hilfe von "SETNAM" (Akku= Länge des Namens, X/

Y= Zeiger auf Namensstring). Anschließend muß "SETPAR" aufgerufen werden. Hierbei kommen Filenummer und Geräteadresse, wie gewohnt, in Akku und X-Register. Im Y-Register wird die Sekundäradresse abgelegt. Jedoch müssen wir hier, anders als sonst, die gewünschte Sekundäradresse plus $60 (= dez.96) übergeben. Dies hat Floppyinterne Gründe. Möchten wir also den Befehlskanal ansprechen, so muß die Sekundäradresse $6F (dez.111) übergeben werden. Möchten wir ein PRG-File lesen, so müssten wir den Wert $60 übergeben, etc. Zuletzt wird wieder wie üblich das File mittels "OPEN" geöffnet.
Soweit also nichts Neues. Nun müssen wir uns, wie vorher auch, verdeutlichen, ob die Floppy senden, oder empfangen soll.
Anstelle von "CHKIN" oder "CKOUT" müssen diesmal jedoch andere Routinen benutzt werden. Möchten wir Daten empfangen, so soll die Floppy mit uns "reden". Deshalb heißt die entsprechende Routine, die sie dazu auffordert uns Daten zuzusenden "TALK" (engl." reden"). Wollen wir Daten senden soll die Floppy uns "zuhören". Die hierfür gedachte Unterroutine heißt "LISTEN" (engl." hören"). Da es, wie oben schon einmal beschrieben, bis zu 6 Geräte geben kann, die uns zuhören oder mit uns reden sollen, muß mit "TALK" oder "LISTEN" auch mitgeteilt werden, welches der Geräte nun für uns bereit sein soll. Hierzu müssen Sie vor Aufruf der Routinen die Geräteadresse des gewünschten Gerätes im Akku ablegen.
Als nächstes muß dem Gerät an der anderen Seite mitgeteilt werden, welcher seiner offenen Kanäle senden oder empfangen soll. Hierzu muß die Sekundäradresse an das Gerät übermittelt werden. Dies geschieht über die Routinen "SECTLK" und "SECLST". Beide verlangen die gewünschte Sekundäradresse (wieder plus $60) als Parameter im Akku. "SECTLK" müssen Sie nach einem "TALK" aufrufen,"SECLST" nach einem "LISTEN". Nun ist alles vorbereitet, um Daten zu lesen oder zu schreiben. Dies erledigen Sie nun am besten mit den beiden speziellen IEC-Ein-/ Ausgaberoutinen, die ich oben schon ansprach. Sie können sie genauso wie "BASIN" und "BASOUT" benutzen. Zum Lesen von Daten benutzen Sie die Routine "IECIN", die das gelesene Byte im Akku zurückgibt. Zum Schreiben benutzen Sie "IECOUT", der das zu schreibende Byte im Akku übergeben werden muß.
Haben Sie die Ein- oder Ausgabe beendet, so müssen Sie das entsprechende Empfangs-, bzw. Sendegerät wieder in den Wartezustand zurückbringen. Dies geschieht über die Routinen "UNTALK" und"UNLIST" . Die erstere müssen Sie nach dem Empfang, die letztere nach dem Senden von Daten benutzen. Dies müssen Sie auch immer dann tun, wenn Sie anschließend auf dem selben Kanal eine andere Operation ausführen wollen (z.B. Befehl an Befehlskanal senden und anschließend Floppystatus auslesen). Wenn Sie mit Ihrer Operation nun fertig sind, so können Sie den zuvor geöffneten Filekanal wieder schließen.
Wie Sie sehen ist diese Methode etwas aufwendiger. Dafür aber können Sie nun problemlos den Befehlskanal öffnen und z.B. den "Memory-Read"-Befehl ("M-R") benutzen. Hierbei öffnen Sie einen Filekanal und teilen der Floppy mit, daß sie Daten empfangen soll ("LISTEN" und "SECLST" aufrufen) . Nun senden Sie mittels "IECOUT" den Befehlsstring. Anschließend setzen Sie die Floppy wieder in der Normalstatus zurück, indem Sie "UNLIST" aufrufen. Hiernach können Sie nun die zuvor verlangten Daten vom Befehlskanal abholen. Senden Sie einfach ein "TALK" und "SECTLK" und lesen Sie die Daten mit "IECIN" ein. Abschließend muß die Floppy wieder mit "UNTALK" in den Wartezustand geschickt werden und wir können das File schließen.
Desweiteren ist das Bedienen von zwei offenen Files nun weitaus besser möglich. Bevor Sie eines dieser Files ansprechen, müssen Sie einfach den entsprechenden Übertragungsmodus festlegen und seine Sekundäradresse an die Floppy schicken. Denken Sie immer daran, daß Sie nach jeder Lese- oder Schreiboperation ein "UNTALK" oder "UNLIST" senden müssen, damit die Floppy auf weitere Befehle reagiert.
Hier nun noch die Einsprung-Adressen der neuen I/ O-Routinen:

LISTEN:	$FFB1
TALK:	$FFB4
SECLST:	$FF93

```
SECTLK:     $FF96
UNLIST:     $FFAE
UNTALK:     $FFAB
IECOUT:     $FFA8
ECIN:       $FFA5
```

PROGRAMM-BEISPIELE

Kommen wir nun noch zu einigen Programmbeispielen, die Ihnen die Benutzung der neuen I/O-Routinen erläutern sollen. Die Routinen aus Beispiel 1 und 2 finden Sie, als Hypra-Ass Quellcode, auf dieser MD unter dem Namen "FK.IO2.S". Der Quellcode für Beispiel 3 steht im File "FK.SHOWBAM.S". Da es sich dabei um ein eigenständiges Programm handelt, existiert hierzu auch noch ein, durch 'RUN' startbares, File namens "FK.SHOW-BAM".

1. LESEN UND SCHREIBEN EINES FILES

Zunächst einmal möchte ich Ihnen zeigen, wie die beiden Routinen "READFILE" und "WRITEFILE" aus dem letzten Kursteil in der neuen Methode aussehen. Hierbei müssen wir lediglich den Aufruf von "CHKIN/ CKOUT" mit dem von "TALK / SECTALK" oder "LISTEN / SECLIST" ersetzen. Der Aufruf von "CLRCH" am Ende der Übertragung muß mit einem "UNTALK" bzw."UNLIST" ersetzt werden. Desweiteren muß den Sekundäradressen beim Öffnen der Wert $60 hinzuaddiert werden ('$60' bei ReadFile,'$61' bei WriteFile). Der Rest des Programms sollte sich von selbst erklären. Damit man die alte und neue Version der Routinen nicht miteinander verwechselt, heißen die neuen Versionen "IECREAD" und "IECWRITE". Die Parameter, die beim Aufruf übergeben werden müssen, sind dieselben wie bei den alten Versionen:

```
IECREAD     stx  $fb            ;Startadresse in
            sty  $fc            ;Zeiger ablegen.
            ldx  #$34           ;Zeiger auf
            ldy  #$03           ; Filenamen
            jsr  setnam         ; setzen.
            lda  #01            ;Fileparameter
            ldx  #08            ; setzen
            ldy  #$60           ; ($60=PRG Lesen)
            jsr  setpar
            jsr  open           ;File öffnen
            lda  #08            ;"Gerät Nr. 8:
            jsr  talk           ; rede bitte!"
            lda  #$60           ;"Und zwar auf Se-
            jsr  sectlk         ; kundäradr. $60)!"
            ldy  #00            ;Offset=0
loop1       jsr  iecin          ;Zeichen lesen
            sta  ($fb),y        ; und ablegen
            nic  $fb            ;Zeiger
            bn   l1             ;um 1
            inc  $fc            ; erhöhen
l1          lda  $90            ;Ende erreicht?
            beq  loop1          ;Nein, dann weiter!
            lda  #08            ; Sonst:" Gerät jsr untalk ; Nr.8 : Ruhe bitte!"
            lda  #01            ; File jmp close ; schließen
;*****************************************
IECWRITE    stx  $fb            ; Startadresse in
            sty  $fc            ; Zeiger ablegen
            ldx  #$34           ;Filenamen
            ldy  #$03           ; setzen
            jsr  setnam
            lda  #01            ;Fileparameter
            ldx  #08            ; setzen
```

```
                ldy  #$61           ; ($61=PRG schr.)
                jsr  setpar
                jsr  open           ;File öffnen
                lda  #08            ;"Gerät Nr8: Hör'
                jsr  listen         ; mir bitte zu!'
                lda  #$61           ;"Und zwar auf Sek.
                jsr  seclst         ; Adr.$61 !")
                ldy  #00            ;Offset=0
loop2           lda  ($fb),y        ;Zeichen holen
                jsr  iecout         ; und abschicken
                inc  $fb            ;Zeiger
                bne  l2             ; um 1
                inc  $fc            ; erhöhen
l2              lda  $fe            ;Und mit
                cmp  $fc            ; Endadresse
                bne  loop2          ; vergleichen
                lda  $fd            ;Wenn ungleich
                cmp  $fb            ; dann weiter
                bne  loop2
                lda  #08            ;"Gerät Nr.8:
                jsr  unlist         ; Bin fertig!"
                lda  #01            ;File
                jmp  close          ; schließen
```

2. FLOPPYSTATUS ANZEIGEN

Ein weiterer Vorteil, der sich aus der Benutzung der direkten IEC-Routinen ergibt, ist, daß Sie nun problemlos, während des Arbeitens mit einem File, Statusmeldungen auf dem Bildschirm ausgeben können. Da die Standard Ein-/ Ausgabe ja nicht verändert wurde, erscheinen Texte, die mit BSOUT ausgegeben werden sollen, auch weiterhin auf dem Bildschirm. Umgekehrt wird bei BASIN von der Tastatur, und nicht etwa aus dem offenen File gelesen.
Als Beispiel für eine solche Anwendung möchte Ich Ihnen das Auslesen des Fehlerkanals der Floppy zeigen. Hierzu muß lediglich der Befehlskanal geöffnet, und die Fehlermeldung, die im ASCII-Code von der Floppy generiert und übermittelt wird, Zeichen für Zeichen ausgelesen und auf dem Bildschirm ausgegeben werden. Ich denke, die Routine erklärt sich von selbst:

```
ERRCHN          lda  #00            ;Länge Filename=0
                jsr  setnam         ;für "Kein Name"
                lda  #01            ;Fileparameter
                ldx  #08            ;setzen
                ldy  #$6f           ;(=Befehlskanal)
                jsr  setpar
                jsr  open           ;Filekanal öffnen
                lda  #08            ;Floppy Nr.8 zum
                jsr  talk           ;Senden auffordern
                lda  #$6f           ;Und zwar auf Kanal
                jsr  sectlk         ;mit Sek.Adr. $6F
ecloop1         jsr  iecin          ;Zeichen lesen
                jsr  bsout          ;Und auf dem Bildschirm ausgeben
                lda  $90            ;Fileende erreicht?
                beq  ecloop1        ;Nein, also weiter
                lda  #08            ;Floppy Nr.8
                jsr  untalk         ;zurücksetzen
                lda  #01            ;Und Befehlskanal
                jmp  close          ;schließen
```

3. BAM ANZEIGEN

Als Nächstes wollen wir ein Programm schreiben, daß uns anzeigt, welche Blocks einer Diskette belegt sind, und welche nicht. Hierzu müssen wir die BAM ("Block Availability Map"),

die im Disk-Header-Block (Track 18 Sektor 0) zu finden ist, auslesen und zu einer grafischen Anzeige dekodieren. Dies ist ein hervorragendes Beispiel um die Benutzung der neuen I/ O-Routinen zu demonstrieren und gleichzeitig zu zeigen, wie man mit der BAM umgeht. Sie erinnern sich vielleicht noch an deren Aufbau.
Ich hatte ihn in Teil 5 dieses Kurses besprochen: Im Disk-Header- Block sind die Bytes 4 bis einschließlich 143 für die BAM reserviert. Hierbei zeigen jeweils 4 Bytes in Folge die Blockbelegung für einen Track an. Im ersten Byte steht dabei die Gesamtanzahl der freien Blocks dieses Tracks. Die Bits des zweiten Bytes kodieren die Belegung der Sektoren 0-7, die Bits des dritten Bytes die Belegung der Sektoren 8-15 und die Bits des vierten Bytes die Belegung der Sektoren 16-23. Hat ein Track weniger Sektoren, als im vierten Byte angezeigt werden, so sind die restlichen Bits dieses Bytes unbenutzt (so z.B. immer die letzten zwei Bits, da es ja nie mehr als 21 Tracks gibt). Ein gesetztes Bit bedeutet, daß der entsprechende Sektor frei ist, ein gelöschtes Bit, daß er belegt ist. Wollen wir einmal klären, welche Dinge zur Lösung des obigen Problems notwendig sind. Zunächst müssen wir den Befehlskanal und anschließend einen Pufferkanal öffnen. Nun muß der Befehl zur Floppy gesandt werden, der den Block 18/0 in den geöffneten Puffer liest. Ist dies geschehen, müssen wir nur noch auf das vierte Byte des Puffers positionieren (dort beginnt ja die BAM) und die vier Belegungs-Bytes für jeden Track einlesen und darstellen. Kommen wir also zu unserem Programm:

```
main    lda  #00            ;Bildschirm-
        sta  53280          ;farben
        lda  #11            ;ein-
        sta  53281          ;stellen
        lda  #<(text1)      ;Text für Anzeige
        ldy  #>(text1)      ;auf Bildschirm
        jsr  strout         ;ausgeben
mloop5  lda  #00            ;Filename setzen
        jsr  setnam         ;(0='Kein Name')
        lda  #01            ;Parameter: LFN 1
        ldx  #08            ;Geräteadresse 8
        ldy  #$6f           ;Sekundäre Adresse 15
        jsr  setpar         ;setzen
        jsr  open           ;Befehlskanal öffnen
        lda  #01            ;Filename "#"
        ldx  #<(bufnam)     ;(für 'Puffer
        ldy  #>(bufnam)     ; reservieren')
        jsr  setnam         ; setzen
        lda  #02            ;Fileparameter
        ldx  #08            ; setzen
        ldy  #$62           ; (Sek.Adr=2)
        jsr  setpar
        jsr  open           Pufferkanal öffnen
```

Bis hier sollte wohl alles klar sein. Wir haben in den obigen Zeilen den Befehlskanal und einen Pufferkanal mit der Sekundäradresse 2 geöffnet (bitte denken Sie daran, daß beim Öffnen und beim Benutzen der Routinen SECTLK und SECLST diese Adresse plus $60 übergeben werden muß).
Als nächstes können wir die beiden Floppybefehle zum Lesen des Blocks 18/0 und zum Positionieren auf Byte 4 senden:

```
        lda  #<(com1)  ;Befehl 'Block
        ldy  #>(com1)  ;18/0 lesen'
        jsr  sendcom           ;senden.
        lda  #<(com2)  ;Befehl 'Puffer-
        ldy  #>(com2)  ;zeiger auf
        jsr  sendcom           ;Byte 4' senden.
```

Wie Sie sehen, benutze ich hier eine eigene Routine, um die Befehle zu senden. Sie heisst

"SENDCOM" und benötigt einen Zeiger auf den Befehlstext in Akku und Y-Register. Zur Beschreibung der Routine kommen wir später. Hier noch die Syntax beiden Befehle, die oben gesandt werden. Sie finden Sie ganz am Ende des Quelltextes:

```
com1        .tx  "u1 2 0 18 0"
            .by  13,0
com2        .tx  "b-p 2 4"
            .by  13,0
```

Wie Sie sehen benutzen wir die Befehle "U1" (die bessere Version von "B-R") und "B-P". Die einzelnen Parameter werden dabei als ASCII-Werte übertragen. Wie Sie auch bemerken, so wird bei den Befehlen die tatsächliche Sekundäradresse, nämlich 2 und nicht etwa $62, benutzt. Der Wert 13 am Ende jedes Befehls wird benötigt, um der Floppy das Ende des Befehlsstrings zu signalisieren. Die 0 wird nicht mitgesandt. Sie dient als Endmarkierung für die SENDCOM-Routine.
Machen wir nun jedoch weiter im Quelltext des Hauptprogramms. Die nun folgenden Zeilen versetzen die Floppy in den Sendestatus und bereiten die Bildschirmausgabe der Tracks vor. Hierbei wird ein Zeiger auf den Bildschirmspeicher in $FB/$FC abgelegt, der nach der Ausgabe eines Sektors um 40 erhöht wird. So erreichen wir eine vertikale Ausgabe des Tracks. Für den nächsten Schleifendurchlauf wird der Zeiger wieder auf den Startwert+1 (nächste Spalte) zurückgesetzt:

```
                lda  #08            ;Floppy Nr. 8 zum
                jsr  talk           ;Senden auffordern
                lda  #$62           ;Und zwar auf

                jsr  sectlk         ;Sekundäre Adresse 2

                lda  #34            ;Trackzähler
                sta  $03            ;initialisieren
                lda  #163           ;Grundwert Zeiger
                sta  mloop3+1       ;einstellen
mloop3 ldx  #163                    ;Bildschirmpo-
                ldy  #04            ;sitionszeiger
                stx  $fb            ;initialisieren
                sty  $fc
                inc  mloop3+1       ;Spalte für nächsten Track um 1 erhöhen
```

Nun erfolgt das eigentliche Lesen und Ausgeben der BAM-Bytes. Hierbei müssen wir immer das jeweils erste Byte überlesen, da es ja keine Blockbelegung ansich, sondern nur die Anzahl der freien Blocks enthält. Die folgenden zwei Bytes können normal ausgegeben werden. Da das letzte Byte immer eine unterschiedliche Anzahl an unbelegten Bits enthält, müssen wir hier aufpassen, daß nur soviele Bits ausgegeben werden, wie tatsächlich benutzt werden. Dies geschieht über die Tabelle "SECTAB", die die Anzahl der benutzen Blocks des letzten Bytes eines jeden Tracks beinhaltet. Hier jedoch erst einmal wieder das Listing:

```
                jsr  iecin          ;1. Byte überlesen
                jsr  iecin          ;2. Byte lesen
                jsr  byteout1       ; und ausgeben
                jsr  iecin          ;3. Byte lesen
                jsr  byteout1       ; und ausgeben
                jsr  iecin          ;4. Byte lesen
                ldy  $03            ;Tracknr. als Zeiger holen
                ldx  sectab,y       ;Anzahl benutzte Bits des Tracks holen.
                jsr  byteout2       ;Und Byte ausgeben
                dec  $03            ;Trackzähler-1
                bpl  mloop3         ;Nochmal, wenn >0
```

Wie Sie sehen, benutze ich eine eigene Routine zur Ausgabe eines BAM-Bytes. Sie heisst "BYTEOUT1" und gibt die Bits 0-7 des BAM-Bytes vertikal auf dem Bildschirm aus. Hierbei wird

ein "o" geschrieben, wenn der Block belegt ist, und ein ".", wenn er unbelegt ist. Die Routine "BYTEOUT2" ist im Prinzip dieselbe, wie BYTEOUT1. Der einzige Unterschied ist, daß die zweite Version die Anzahl der auszugebenden Bits im X-Register verlangt. Diese wird nun über die SECTAB-Tabelle eingelesen. Achten Sie darauf, daß die Liste rückwärts gelesen wird, und somit der Reihe nach die Blockanzahlen der Tracks 35-1 (jeweils nur für das letzte Byte und minus 1) in der Liste verzeichnet sind. Sie finden die Liste ebenfalls am Ende des Quellcodes.
Fahren wir nun weiter im Hauptprogramm.
Nachdem alle BAM-Bytes gelesen und ausgegeben wurden, können wir Befehls und Pufferkanal wieder schließen. Vorher muß die Floppy allerdings wieder in den Wartezustand zurückversetzt werden. Hierauf folgt noch eine Tastenabfrage, die das Programm bei Tastendruck wiederholt und bei der Taste '<' den Bildschirm löscht und zurückspringt:

```
                lda  #08          ;Floppy wieder
                jsr  untalk       ; zurücksetzen.
                lda  #02          ;Pufferkanal
                jsr  close        ; schließen
                lda  #01          ;Befehlskanal
                jsr  close        ; schließen
mloop4          jsr  inkey        ;Taste holen
                beq  mloop4       ; Keine --> weiter
                cmp  #95          ;= '<' ?
                beq  end          ;Ja, also ENDE.
                jmp  mloop5       ;Nein,also nochmal
end             jmp  $e544
```

Wollen wir uns nun die Routine SENDCOM anschauen. Im Prinzip ist sie nichts anderes als eine STROUT-Routine, nur daß die Ausgabe nicht auf dem Bildschirm, sondern an die Floppy erfolgt. Sie finden sie in den Zeilen 990-1070 des Quelltextes:

```
sendcom         sta  $fb          ;Zeiger auf String
                sty  $fc          ;ablegen
                lda  #08          ;Floppy Nr.8 zum
                jsr  listen       ; Empfang auffordern
                lda  #$6f         ;Und zwar von Sek.
                jsr  seclst       ;Adresse 15
                ldy  #00          ;Offset=0
scloop1         lda  ($fb),y      ;Zeichen lesen
                beq  s1           ;Wenn 0 --> fertig
                jsr  iecout       ;Zeichen senden
                iny               ;Zeiger um 1
                bne  scloop1      ;erhöhen und
                inc  $fc          ;Schleife wieder-
                jmp  scloop1      ;holen
s1              lda  #08          ;Ende. Floppy Nr.
                jmp unlist        ;8 zurücksetzen
```

Das soll es dann wieder einmal für diesen Monat gewesen sein. Sie sind nun komplett in die Bedienung der Floppy 1541 eingeführt und sollten eigentlich auch schon eigene Programme schreiben können. Trotzdem wird es nächsten Monat noch einen Teil dieses Floppy-Kurses geben, in dem wir ein ganz besonderes Beispielprogramm besprechen werden wollen.

(ub)

Teil 9 – Magic Disk 02/93

Herzlich Willkommen zum neunten und letzten Teil des Floppy-Kurses. In den vorangehenden Folgen haben wir einiges über den Aufbau von Disketten und das Ansprechen der Floppy

gelernt. Ich möchte diesen Kurs nun mit einem anspruchsvollen Programmbeispiel abschließen, das wir uns im Laufe der heutigen Folge erarbeiten wollen. Es dient als Beispiel für die Programmierung der Floppy in Assembler und gleichzeitig auch als Beispiel für das was möglich ist, wenn eine Diskette richtig manipuliert wird.

DIE AUFGABE

Sicherlich kennen Sie das Problem: Sie sind gerade dabei Ihre Diskettensammlung aufzuräumen und Ihre Programme so umzukopieren, daß jede Diskette optimale Platzausnutzung aufweisen soll. Am Besten so, daß restlos jeder Block dieser Diskette beschrieben ist. Beim Zusammenstellen bleiben dann aber noch noch 17 Blocks übrig - Mist, kein Platz mehr für das letzte,30 Blocks lange Programm!
Was tun? Im Prinzip bleiben Ihnen nur die folgenden Möglichkeiten:

1. Sie fangen nochmal von vorne an (Gähn - Kopieren dauert lange).
2. Sie belassen alles, wie es ist (Ärgerlich -17 Blocks verschenkt).
3. Sie packen das letzte Programm (Wieder Gähn - dauert auch lange und ausserdem hat das File hinterher bestimmt genau 18 Blocks, so daß es doch nicht passt).
4. Sie benutzen das Programm "USEDIR", das wir uns in diesem Kursteil erarbeiten wollen.

Nun werden Sie fragen:"Na schön, und wie will dieses Programm nun 13 weitere Blocks freibekommen, wenn die Diskette voll ist?". Ganz einfach: aus dem Directory."USEDIR" nimmt sich die Tatsache zunutze, daß der Track 18, in dem das Directory einer Diskette steht, nicht für normale Files zur Verfügung steht.
Er ist lediglich für die Fileeinträge reserviert. In jedem der 18 Fileeintragsblocks (Sektoren 1-18,0 ist für DiskHeader und BAM reserviert) können nun 8 Einträge stehen, was einer maximal mögliche Anzahl von 144 Einträgen entspricht. Da eine normale Diskette aber so gut wie nie so viele Files beherbergt, liegen eine Menge dieser Directoryblocks brach. Gerade bei einer randvollen Diskette werden sie bis in alle Ewigkeit unbenutzt bleiben, da ja keine neuen Files mehr hinzukommen können. Im günstigsten Fall, nämlich dann, wenn Sie weniger als 9 Files auf der Diskette haben, sind das 17 Blocks, die noch zusätzlich frei sind! Diese Blocks soll UseDir nun einfach als Datenblocks verwenden. Es muß lediglich einen Fileeintrag kreieren, in dem der Zeiger für den ersten Track und Sektor auf einen der unbenutzten DirBlocks zeigt. Wird dieses File dann geladen, so greift das DOS der Floppy schön brav auf diese sonst ungenutzten Blocks zu, so als gäbe es keinen Unterschied. Tatsächlich hat die Diskette dann jedoch 683 Blocks (maximal) anstelle von nur 664 !

DIE THEORIE

Was muß unser Programm nun tun, um ein File auf die oben beschriebene Weise umzukopieren. Zunächst einmal wollen wir hier eine Einschränkung vereinbaren, die uns die Arbeit erleichtern soll: ein File, das auf diese Weise installiert werden soll, muß länger sein, als es freie Directoryblöcke gibt. Daraus ergibt sich natürlich, daß die Diskette mindestens noch einen 'normalen', freien Datenblock hat. Das zu installierende File muß desweiteren natürlich kleiner, oder gleich lang der insgesamt verfügbaren Blöcke sein und sollte vom Typ "PRG" sein. Daraus ergeben sich folgende Aufgaben für unser Programm:

1. Das zu installierende File einlesen und benötigte Anzahl von Blöcken ermitteln.
2. Anzahl der freien Blöcke der Zieldiskette ermitteln.
3. Anzahl der freien Directoryblöcke der Zieldiskette ermitteln.
4. Vergleichen, ob das File den obig gegebenen Restriktionen entspricht und ob noch genügend Platz auf der Zieldiskette ist.
5. Wenn ja, dann weiter, sonst abbrechen.
6. Nun muß berechnet werden, wieviele Bytes in den Directoryblocks Platz haben. Alles restliche wird anschließend, als normales File, mit Hilfe der WriteFile-Routine aus dem vorletzten Kursteil auf die Diskette geschrieben.

7. Jetzt suchen wir den Fileeintrag des soeben gespeicherten Files aus den Directoryblocks heraus und Lesen die Informaton für den ersten Track und Sektor aus dem Eintrag aus.
8. Diese Information wird sogleich in die Trackund Sektornummer des ersten freien Directoryblocks abgeändert.
9. Jetzt müssen wir nur noch die fehlenden Directoryblocks schreiben und den letzten Directoryblock auf den ersten Track / Sektor des geschriebenen Files zeigen lassen - Fertig ist die Installation!

DIE BENÖTIGTEN PROGRAMMTEILE

Kommen wir nun also zu unserem Programm. Dabei möchte ich Ihnen zunächst einmal eine Liste der darin enthaltenen Routinen geben. Hierbei habe ich in zwei Arten unterschieden: in Steuer- und IO-Routinen. Diese beiden Namen treffen Ihre Bedeutung zwar nicht voll und ganz, jedoch musste irgendwo eine Grenze gezogen werden.
Die erste Art, die Steuerroutinen also, sind alles Funktionen, die entweder nichts direkt mit der Floppy zu tun haben, und somit für uns uninteressant sind, oder aber Funktionen, die wir in vorangegangenen Kursteilen schon besprochen hatten, und somit nicht noch einer zweiten Dokumentation bedürfen. All diese Routinen werden nur mit ihrem Namen, ihrer Funktion und ihren Parametern aufgeführt, damit Sie wissen, wozu sie da sind, und wie man sie benutzt.
Die zweite Art von Routinen sind größtenteils Floppy-Routinen, die eine Beschreibung benötigen und im Laufe dieses Kursteils alle noch einmal genauer dokumentiert sind.
Desweiteren werden diverse Speicherzellen als Zwischenspeicher für verschiedentliche Werte benutzt. Diese haben, der besseren Übersichlichkeit wegen, richtige Namen, und können im Prinzip überall im Speicher stehen. Dennoch will ich die von mir benutzten Speicherzellen hier einmal aufführen. Sie liegen, bis auf einige Außnahmen, im Speicherbereich von $0332-$03FF, dem Kasettenpuffer also, der bei Floppybenutzung ja unbenutzt, und deshalb verwendbar ist. Hier nun also die besprochenen Beschreibungen:

1. BENUTZTE SPEICHERZELLEN:

- MEM0-MEM4
  Die fünf Zeropageadressen von $02 bis $06. Sie werden für die Zwischenspeicherung verschiedenster Ergebnisse herangezogen.
- FILEMEM ($0F00)
  Dies ist die Anfangsadresse des Speichers, in den das zu installierende Programm geladen wird.
- NUMBUFF ($0332)
  Hier wird der ASCII-String einer Konvertierten Zahl abgelegt (maximal 5 Zeichen).
- NEED ($0337)
  Zwischenspeicher für die Länge des gelesenen Files in Blocks.
- DSKFREE ($0338/$0339)
  Speicher für die Anzahl der freien Blocks der Zieldiskette.
- ALLFREE ($033A/$033B)
  Anzahl der insgesamt (inkl. freie Dir-blocks) auf der Zieldiskette verfügbaren Blocks.
- DIRFREE ($033C)
  Anzahl der freien Directoryblocks
- DIRSTRT ($033D)
  Sektornummer des ersten freien Directoryblocks.
- FTRACK ($033E)
  Tracknummer des ersten, vom normal geschriebenen File benutzten, Datenblocks.

- FSECTOR ($033F)
  Sektornummer des ersten, vom normal geschriebenen File benutzten Datenblocks.
- DSINDEX ($0340)
  Indexzeiger auf den aktuellen Eintrag der Dirblockliste.
- COMBUFF ($0341-)
  Zwischenspeicher für Befehle, die an die Floppy gesandt werden sollen.
- NAME
  Dieses Label bezeichnet die Startadresse, des 17 Zeichen langen Zwischenspeichers für Filenamen. Dieser befindet sich direkt im Sourcecode.

2. STEUERROUTINEN

- GETIN
  Diese Funktion verlangt keine Parameter. Sie liest mit Hilfe der BASIN-Routine des Betriebssystems einen maximal 16 Zeichen langen String von der Tastatur ein und wird zum Eingeben des Filenamens benutzt. Dieser String liegt nach dem Rücksprung ab der Adresse NAME. Die Länge des Filenamens steht in der Speicherzelle MEM0. Am Ende des Namensstring wird ein Nullbyte als Endmarkierung angefügt.
- STROUT: Diese Routine gibt einen ASCII- Text auf dem Bildschirm aus. Lo-/-Hi-Byte der Textadresse müssen in Akku und Y-Register übergeben werden. Der Text muß mit einem Nullbyte enden.
- OPENCOM
  Diese Routine öffnet den Befehlskanal mit der logischen Filenummer 1. Beim Öffnen wird die Diskette automatisch initialisiert (Floppybefehl "I")
- OPENIO
  Hier wird der Befehlskanal mit der Filenummer 1 (wie OPENCOM) und ein Pufferkanal mit der Filenummer 2 (und Sekundäradresse 2) geöffnet.
- RFILE
  Dies ist die "ReadFile"-Routine aus dem letzten Kursteil. Sie liest das File, dessen Namen bei NAME steht und dessen Namenslänge in MEM0 abgelegt ist, an die Adresse FILEMEM. Sie benutzt die Zeropageadressen $F9/$FA als Lesezeiger. Hier steht nach dem Rücksprung gleichzeitig auch die Endadresse des gelesenen Files.
- WFILE
  Schreibt das File in NAME und MEM0 auf Diskette. Die Startadresse des zu speichernden Bereichs muß dabei in den beiden Zeropageadressen $FD/$FE, die Endadresse in $ F9/$FA stehen.
- STATOUT
  Gibt die ermittelten Werte für "Anzahl benötigte Blocks","Freie DirBlocks", und "Freie Diskblocks" auf dem Bildschirm aus.

3. IO-ROUTINEN:

- STRIEC
  Wie "STROUT", nur daß diesmal ein String auf den IEC-Bus (also an die Floppy) ausgegeben wird.
- SENDCOM
  Da wir zur Erfüllung unserer Aufgabe immer nur Floppybefehle benötigen, die aus einem festen String und einer angehängten, variablen Zahl bestehen, wird diese Routine

benutzt, um einen Befehl, dessen String an der Adresse in X- und Y-Register steht und dessen abschließende Nummer im Akku übergeben wurde, an die Floppy zu senden.

- WDUMMY
  Diese Routine legt auf der Zieldiskette ein File mit dem Namen in NAME und MEM0 an und löscht es direkt wieder. Wozu dies notwendig ist, werden wir später sehen.
- GETDSKF
  Ermittelt die Anzahl der freien Blocks der Zieldiskette und legt sie in DSKFREE ab.
- GETDIRF
  Ermittelt die Anzahl der freien Directoryblocks der Zieldiskette und legt sie in DIRFREE ab. Desweiteren wird die Summe von DSKFREE und DIRFREE berechnet und in ALLFREE abgelegt.
- GETNEED
  Berechnet die benötigte Anzahl Blöcke des zuvor eingelesenen Files und legt Sie bei " NEED" ab.
- CHGENT
  Diese Routine sucht den Filenamen in NAME aus dem Directory heraus, liest die Nummern des ersten Tracks und Sektors dieses Files ein, und überschreibt diese beiden Informationen mit den Werten für den ersten freien Directoryblock.
- WBLOCKS
  Diese Routine schreibt alle fehlenden Blocks des Files in die freien Directoryblocks und gibt im letzten dieser Blocks Track- und Sektornummer des ersten Datenblocks des ' normal' gespeicherten Files an.

DIE PRAXIS

Nach all der trockenen Theorie wollen wir nun endlich zur Praxis schreiten. Beginnen möchte ich mit der Steuerroutine 'MAIN' unseres Programms, in der das oben aufgeführte Aufgabenschema in Programmcode umgesetzt ist. Anschließend wollen wir uns mit den benutzten Unterroutinen beschäftigen.

1.MAIN

Diese Routine steuert das gesamte Programm. Zunächst einmal wollen wir die Bildschirmfarben setzen, den Titeltext ausgeben und den Namen des zu installierenden Files ermitteln. Ist dieser Name gleich dem String X", so soll das Programm mit einem RESET verlassen werden:

```
main        lda  #11            ;Bildschirm-
            sta  53280          ;farben
            sta  53281          ;setzen.
            lda  #<(text1)      ;Titeltext
            ldy  #>(text1)      ;auf Bildschirm
            jsr  strout         ;ausgeben.
            Jsr  getin          ;Und Filename einlesen.
            cpy  #1             ;Vergleichen, ob
            bne  m3             ;der Name="X"
            lda  #"x"           ;ist.
            cmp  name           ;Wenn nein, dann
            bne  m3             ;weitermachen.
            jmp  64738          ;Sonst: RESET.
```

Als Nächstes müssen wir das File lesen und seine Blocklänge berechnen. Hieraufhin wird der Benutzer dazu aufgefordert, die Zieldiskette einzulegen, von der wir dann die Anzahl der freien

Dir- und Diskblocks ermitteln. Gleichzeitig wird dann auch noch das Dummyfile erzeugt. Am Ende werden alle ermittelten Werte mittels der Routine STATOUT auf dem Bildschirm ausgegeben:

```
m3      jsr  rfile          ;File einlesen
        jsr  getneed        ;Anzahl der benötigten Blocks berechnen
        lda  #<(text2)      ;"Zieldisk ein-
        ldy  #>(text2)      ;legen"
        jsr  strout         ;ausgeben und
mloop1  jsr  inkey          ;auf Tastendruck
        beq  mloop1         ;warten.
        lda  #<(text3)      ;"Untersuch
        ldy  #>(text3)      ;Zieldisk"
        jsr  strout         ;ausgeben.
        jsr  wdummy         ;Dummy-File anlegen und löschen.
        jsr  openio         ;Kanäle öffnen
        jsr  getdskf        ;Freie Diskblocks ermitteln
        jsr  getdirf        ;Freie Dirblocks ermitteln.
        jsr  closeio        ;Kanäle schließen
        jsr  statout        ;Werte ausgeben.
```

Nachdem nun all diese Dinge getan sind, müssen wir nun erst einmal prüfen, ob das gelesene File und die Daten der Zieldiskette es uns ermöglichen, das File auf die Diskette zu schreiben. Hierbei wird abgebrochen, wenn keine Dirblocks mehr frei sind, das File kürzer als die noch verfügbaren Dirblocks, oder länger als die gesamt verfügbaren Blocks ist:

```
        lda  dirfree        ;DirFree lesen
        bne  m1             ;wenn<>0, weiter.
        lda  #<(errtxt1)    ;Sonst "Kein Dir-
        ldy  #>(errtxt1)    ;blk mehr frei"
        jmp  errout         ;ausg. u. Ende.
m1      cmp  need           ;DirFree mit NEED
        bcc  m2             ;vergleichen.
        lda  #<(errtxt2)    ;Wenn >, dann
        ldy  #>(errtxt2)    ;"File zu kurz"
        jmp  errout         ;ausg. u. Ende.
m2      ldy  allfree+1      ;Hi-Byte ALLFREE lesen.
        bne  ok             ;Wenn <>0, dann genug Platz.
        lda  allfree+0      ;Sonst Lo-Byte lesen
        cmp  need           ;u. m. NEED vgl.
        bcs  ok             ;Wenn >, Ok.
        lda  #<(errtxt3)    ;Sonst "File zu
        ldy  #>(errtxt3)    ;lang" aus-
        jmp  errout         ;geben
```

Wurden all diese Vergleiche erfolgreich bestanden, so kann das File installiert werden. Hierzu müssen zunächst einige Vorbereitungen getroffen werden: Als erstes sollten wir die Sektornummer des ersten freien Directoryblocks ermitteln.
Dies ginge natürlich dadurch, indem wir die BAM einlesen würden, und uns einen unbelegten Block als Startblock heraussuchen würden. Es geht aber noch viel einfacher: das DOS der Floppy beschreibt die Sektoren einer Diskette nämlich meistens in einer ganz bestimmten Reihenfolge. Das ist bei den normalen Datenblocks nicht unbedingt gesichert, da bei einer nahezu vollen Diskette diese Reihenfolge nicht mehr eingehalten werden kann. Beim Directorytrack 18 können wir uns jedoch mit 100%-iger Sicherheit darauf verlassen, daß sie immer eingehalten wird. So gibt es nämlich eine ganz bestimmte Reihenfolge, in der die Directorytracks geschrieben werden. Wir müssen lediglich wissen, wieviele Blocks benutzt sind, und uns die Sektornummer des nächsten Blocks aus einer Tabelle zu holen. Dies ist dann automatisch der erste leere Dirblock. Die angesprochene Tabelle ist bei dem Label "BLKTAB"

abgelegt, und beinhaltet die folgenden Werte für Sektornummern:

```
BLKTAB      .byte 0,1,4,7,10,13,16
            .byte 2,5,8,11,14,17
            .byte 3,6,9,12,15,18,$ff
```

In dieser Reihenfolge werden die Blocks des Track 18 IMMER belegt. Der Wert $FF am Ende der Liste steht hier als Endmarkierung der Liste, die später beim Beschreiben der Dirblocks von Bedeutung ist. Nun müssen wir erst einmal die Sektornummer des ersten freien Dirblocks ermitteln:

```
ok          lda #19             ;Von der Gesamtsektorzahl (19) für Track 18
            sec                 ;die Anzahl der
            sbc dirfree         ;freien Dirblocks subtrahieren (=Anzahl belegte Blocks) und als
            tay                 ;Index in Y-Reg.
            lda blktab,y        ;Sektornr. lesen
            sta dirstrt         ;und speichern.
            sty dsindex         ;Index speichern
```

Nun wollen wir das File auf der Zieldiskette anlegen. Hierzu schreiben wir es zunächst ganz normal mittels der WFILE-Routine. Hierbei sollen jedoch nur die Bytes geschrieben werden, die nicht mehr in die Dirblocks passen. Wir müssen nun also die Anfangsadresse des zu schreibenden Files berechnen. Da die ersten beiden Bytes eines Datenblocks immer als Zeiger auf den nächsten Datenblock dienen, müssen wir also den Wert DIR-FREE*254 zur FILEMEM-Adresse hinzuaddieren, um unsere Anfangsadresse zu erhalten:

```
            ldx dirfree         ;DIRFREE in X laden (=HiByte Startadr.).
            txa                 ;u. in Akku holen
            dex                 ;HiByte-1
            asl                 ;Akku*2
            eor #$ff      ;und invertieren
            clc                 ;mit
            adc #1              ;Übertrag.
            adc #<(filemem)     ;LoByte von
            bcc m5              ;FILEMEM
            inx                 ;addieren.
m5          sta $fd             ;unde ablegen
            txa                 ;HiByte holen
            clc                 ;und mit HiByte
            adc #>(filemem)     ;FILEMEM add.
            sta $fe             ;u. ablegen
            jsr wfile           ;File schreiben
```

Nachdem die Anfangsadresse berechnet und in $ FD/$FE abgelegt wurde, wird die WFILE-Routine aufgerufen. Die Endadresse muß in $F9/$FA stehen, wo sie noch von der RFILE-Routine enthalten ist (wird von ihr als Lesezeiger verwendet).
Ist das File nun geschrieben, so müssen wir nur noch seinen Fileeintrag aus dem Directory heraussuchen, Track und Sektor des ersten freien Dirblocks eintragen und die fehlenden DIRFREE*254 anfänglichen Bytes in den Dirblocks unterzubringen:

```
            jsr  openio         ;Kanäle öffnen
            jsr  chgent         ;Eintrag suchen und ändern.
            jsr  wblocks        ;DirBlocks schreiben.
            jsr  closeio        ;Kanäle schließen
            lda  #<( text4)     ;" File install
            ldy  #>( text4)     ; liert"
errout      jsr  strout         ;ausgeben.
eo1         jsr  inkey          ;Auf Tastendruck
            beq  eo1            ;warten.
```

```
                jmp  main           ;Und neu starten
```

Soviel also zu unserer Steuerroutine. Kommen wir nun zu den Unterfunktionen:

2. STRIEC

Diese Routine gibt einen Zeichenstring, dessen Adresse in Akku und Y-Register steht, an den Befehlskanal aus. Der String muß mit einem Nullbyte beendet sein:

```
striec          sta  $fb            ; Adresse in Zei
                sty  $fc            ; ger ablegen.
                Lda  #8             ;Floppy auf
                jsr  listen         ;Befehlskanal
                lda  #$6f           ;empfangsbereit
                jsr  seclst         ;machen.
                Ldy  #0             ;String lesen
siloop1         lda  ($fb),y        ;und senden.
                bne  si1
                lda  #8             ;Floppy zurück-
                jmp  unlist         ;setzen u. Ende.
si1             jsr  iecout         ;Zeichen senden
                iny                 ;und Zeiger+1
                bne  siloop1
                inc  $fc
                jmp  siloop1
```

3. SENDCOM

Diese Routine wird dazu verwandt, einen Floppybefehl zu senden. Hierbei unterscheidet sie sich jedoch von der Routine STRIEC. Es wird nämlich nicht nur der Zeiger des Befehls übergeben (in X- und Y-Register), sondern auch ein Wert im Akku, der vor dem Senden in ASCII umgewandelt und an den Befehlsstring in X/ Y angehängt wird. Der gesamte Befehl wird dabei bei COMBUFF abgelegt und dann mittels STRIEC an die Floppy gesandt:

```
sendcom         pha                 ; Akku retten
                stx  scloop1+1      ; Stringzeiger
                sty scloop1+2       ;setzen
                ldy  #0             ;Und String
scloop1         lda  $c000,y        ;nach COMBUFF
                beq  sc1            ;umkopieren.
                sta  combuff,y
                iny
                jmp  scloop1
sc1             sty  mem0           ;Zeiger retten,
                pla                 ;Akku holen,
                jsr  i2a            ;konvertieren.
                ldy  mem0           ;Zeiger zurückh.
                Ldx  #0             ;und in ASCII
scloop2         lda  numbuff,x      ;konvertierte
                beq  sc2            ;Zahl an COMBUFF
                sta  combuff,y      ;anhängen.
                inx
                iny
                jmp  scloop2
sc2             lda  #13            ;CR anhängen
                sta  combuff,y
                iny
                lda  #0             ;Endmarkierung
                sta  combuff,y      ;anhängen
                lda  #<( combuff)   ;und Inhalt von
                ldy  #>( combuff)   ;COMBUFF an
```

```
                jmp  striec             ;Floppy senden.
```

4. WDUMMY

Kommen wir nun zur Routine "WDUMMY". Wir sollten zunächst einmal klären, wozu sie benötigt wird. Wie Sie oben sahen, wird sie aufgerufen, noch BEVOR irgendetwas anderes getan wird. Hierbei tut sie eigentlich nichts anderes, als ein File mit dem Namen in NAME und MEM0 auf der Zieldiskette anzulegen und gleich darauf wieder zu löschen. Das deshalb notwendig, um sicherzustellen, daß auch die richtige Anzahl an Dirblocks als 'belegt' gekennzeichnet ist. Sollten nämlich genau 8 (oder 16,24, etc.) Files auf der Zieldiskette vorhanden sein, so kann es zu Problemen kommen. Beim Schreiben eines neuen Files muß das DOS dann nämlich einen neuen Dirblock hinzufügen, der ab dann nicht mehr zur Datenspeicherung benutzt werden kann. Damit es dabei keine Konfrontationen gibt, wird das File also sporadisch schon einmal angelegt und direkt danach wieder gelöscht. Der neue DirBlock wird dann nicht wieder freigegeben. Der Eintrag bleibt nämlich erhalten, es wird lediglich der Filetyp 'DEL' an das Programm vergeben. Hier nun also die Routine:

```
wdummy      lda  mem0           ;Filename
            ldx  #<(name)       ;in NAME
            ldy  #>(name)       ;und MEM0
            jsr  setnam         ;setzen.
            Lda  #1             ;Fileparameter
            ldx  #8             ;"1,8,1" für
            ldy  #1             ;("PRG saven")
            jsr  setpar         ;setzen.
            jsr  open           ;File öffnen.
            Ldx  #1             ;Kanal 1 als
            jsr  ckout          ;Ausgabefile
            jsr  bsout          ;Byte ausgeben.
            jsr  clrch          ;Standardkanäle zurücksetzen.
            Lda  #1             ;File wieder
            jsr  close          ;schließen.
            jsr  opencom        ;Befehlskanal öffnen
            lda  #<(name-2)     ;Name-2 als
            ldy  #>(name-2)     ;Adresszeiger
            jsr  striec         ;senden.
            Lda  #1             ;Befehlskanal
            jmp  close          ;schließen.
```

Nun kann ich Ihnen auch den Grund zeigen, warum der Filename im Sourcecode abgelegt wird. Hier ist nämlich vor dem eigentlichen Namen auch noch der Text "S:" abgelegt. Wenn wir nun NAME-2 an STRIEC übergeben, so entspricht das einem Scratchbefehl für den Filenamen ("S: NAME"). Da STRIEC ein Nullbyte als Endmarkierung verlangt, wurde die GETIN-Routine so ausgelegt, daß sie nach dem letzten eingelesenen Zeichen ein solches Byte in den NAME-Puffer schreibt. Hier nun die Source-Definition des Filenamenspuffers. Für den Namen werden 17 Bytes reserviert, da ja maximal 16 Zeichen plus die Endmarkierung 0 vonnöten sind:

```
.text "s:"name
.byte 0,0,0,0,0,0,0,0
.byte 0,0,0,0,0,0,0,0,0
```

5. GETNEED

Diese Routine wird benutzt, um die Anzahl der vom Quellfile benötigten Blocks zu ermitteln. Hierbei wird zunächst die Startadresse des Filespeichers subtrahiert, um die effektive Länge in Bytes zu ermitteln. Hiernach wird das High-Byte mit zwei multipliziert. Dies ist nämlich die Anzahl Bytes, die Aufgrund des Wegfalls der ersten beiden Bytes eines Datenblocks zu der Anzahl Lo-Bytes addiert werden muß. Ist dieser Wert größer als 256, so wird ein Block mehr gebraucht. Jetzt wird noch die Anzahl der Low-Bytes hinzuaddiert. Gibt es auch hier einen

Überlauf, so muß wieder ein Block hinzuaddiert werden. Letztendlich muß wieder ein Block hinzugefügt werden, da die letzten Bytes, selbst wenn es weniger als 254 sind, dennoch einen ganzen Block belegen:

```
getneed     ldx  #0              ;NEED
            stx  need            ;löschen.
            lda  $f9             ;Endadresse des Quellfiles (von
            ldx  $fa             ;RFILE noch da) lesen.
            Sec                  ;und Startadresse
            sbc  #<(filemem)     ;subtrahieren.
            bcs  rf1             ;Ergebnis wird in
            dex                  ;MEM1 (LoByte)
rf1         sta  mem1            ;und
            txa                  ;MEM2 (HiByte)
            sec                  ;abgelegt.
            sbc  #>(filemem)
            sta  mem2
            rol                  ;HiByte*2
            bcc  cn1             ;Wenn<256, weiter
            inc  need            ;Sonst Blocks+1
cn1         clc                  ;LoByte addieren
            adc  mem1
            beq  cn2
            bcc  cn2
            inc  need            ;Bei Überlauf Blocks+1
cn2         inc  need            ;Und nochmal Blocks+1
            lda  mem2            ;Und Hi-Byte
            clc                  ;addieren.
            adc  need            ;und in NEED
cn3         sta  need            ;ablegen.
            rts
```

6. GETDISKF

Kommen wir nun zur Routine zum Feststellen der freien Blocks der Diskette. Dieser Wert ist normalerweise nicht direkt aus dem DiskHeaderBlock zu erfahren. Die einzige Möglichkeit wäre, die Blockbelegungen der einzelnen Tracks aus der BAM zu lesen und aufzusummieren. Aber auch hier wollen wir uns eines kleinen Tricks bedienen. Wird eine Diskette nämlich initialisiert (was wir beim Öffnen des Befehlskanals schon tun), so liest die Floppy die BAM der Diskette automatisch ein und berechnet die Anzahl der freien Blocks von selbst. Diese Anzahl wird dann in den Bytes $02FA (Lo) und $02FC (Hi) des Floppyspeichers abgelegt. Was liegt also näher, als diesen Wert direkt, über den Memory-Read- Befehl der Floppy auszulesen. Und nichts anderes tut GETDISKF:

```
getdskf     lda  #<(com5)        ;Memory-Read
            ldy  #>(com5)        ;Befehl
            jsr  striec          ;senden.
            Lda  #8              ;Floppy sende-
            jsr  talk            ;bereit auf
            lda  #$6f            ;Befehlskanal
            jsr  sectlk          ;machen.
            jsr  iecin           ;Lo-Byte lesen
            sta  dskfree+0       ;und sichern.
            jsr  iecin           ;Byte überlesen.
            jsr  iecin           ;Hi-Byte lesen
            sta  dskfree+1       ;und sichern.
            Lda  #8              ;Floppy zurück-
            jmp  untalk          ;setzen.
```

Zusätzlich hierzu muß noch das entsprechende Memory-Read- Kommando im Sourcetext abgelegt werden (3 Zeichen ab Adresse $02FA lesen):

```
com5        .text "m-r"
            .byte 250,2,3,13,0
```

7. GETDIRF

Nun kommen wir zur Routine zum Ermitteln der freien Directoryblocks. Hier können wir keinen Umweg gehen, sondern müssen die BAM direkt auslesen. Da das Directory ausschließlich in Block 18 steht, genügt es, das erste Byte des BAM-Eintrags für Track 18 auszulesen. Dieses Byte steht an Position 72 des DiskHeaderBlocks. Wir müssen also den Block 18/0 in den Puffer lesen, und den Pufferzeiger auf 72 setzen um an die gewünschte Information zu kommen.
Gleichzeitig berechnet diese Routine die Anzahl der insgesamt verfügbaren Blocks auf Diskette. Hierzu müssen wir lediglich den Inhalt von DSKFREE und DIRFREE addieren und in ALLFREE ablegen.

```
getdirf     lda  #<(com6)       ;"Block 18/0
            ldy  #>(com6)       ;lesen"
            jsr  striec         ;senden.
            lda  #<(com7)       ;"Pufferzeiger
            ldy  #>(com7)       ;auf Byte 72"
            jsr  striec         ;senden.
            Lda  #8             ;Floppy sende-
            jsr  talk           ;bereit auf
            lda  #$62           ;Pufferkanal
            jsr  sectlk         ;machen
            jsr  iecin          ;Wert lesen
            sta  dirfree        ;und sichern.
            ldx  dskfree+1      ;Hi-Byte lesen
            clc                 ;Lo-Byte zu
            adc  dskfree+0      ;DIRFREE
            bcc  gf1            ;addieren.
            inx
gf1         sta  allfree+0      ;und in ALLFREE
            stx  allfree+1      ; ablegen.
            Lda  #8             ;und Floppy
            jmp  untalk         ;zurücksetzen.
```

Auch hier benötigen wir zwei Sourcecodetexte für die Diskkommandos:

```
com6        .text"u1 2 0 18 0"  ;Sekt. 18/0
            .byte 13,0          ;lesen
com7        .text "b-p 2 72"    ;Pufferzgr.
            .byte 13,0          ;auf 72.
```

8. CHGENT

Kommen wir nun zu einer der wichtigsten Routinen unseres Programms. Sie durchsucht die Directoryblocks nach unserem Fileeintrag, liest aus ihm den Track/ Sektor des ersten 'normalen' Datenblocks aus und legt ihn in den Adressen FTRACK und FSEKTOR ab. Desweiteren wird hier dann Track und Sektor des ersten, freien Directoryblocks eingetragen und der Block wieder auf die Diskette zurückgeschrieben. Auch diese Routine benutzt die Sektorentabelle BLKTAB, um die Reihenfolge der Directroyblocks zu ermitteln. Desweiteren benötigt sie noch eine weitere Tabelle, in der die Anfangspositionen der einzelnen Fileeinträge eines Directoryblocks abgelegt sind. Diese Tabelle heißt ENTTAB und sieht folgendermaßen aus:

```
enttab      .byte 2,34,66,98,130,162
            .byte 194,226
```

Kommen wir nun jedoch zur eigentlichen Routine. Sie besteht im Prinzip aus zwei ineinander verschachtelten Schleifen, die nacheinander die einzelnen Dirblocks einlesen und jeden einzelnen Eintrag mit dem Namen in NAME vergleichen. Wird der Name gefunden, so werden die Schleifen verlassen. Dann wird nochmals auf diesen Eintrag positioniert, und zwar so, daß der Bufferpointer direkt auf die zwei Bytes für Starttrack und -sektor zeigt und dort die Werte für den ersten freien Dirblock eingetragen (Track 18, Sektor in DIRSTRT) . Abschließend wird dieser Directoryblock inklusive der Änderung wieder auf die Diskette zurückgeschrieben:

```
chgent      lda  #<(text6)      ;"Suche
            ldy  #>(text6)      ;Eintrag"
            jsr  strout         ;ausgeben
            lda  #1             ;Blockzähler
            sta  mem1           ;initialisieren
celoop1     lda  #8             ;Floppy zurück-
            jsr  untalk         ;setzen.
            ldy  mem1           ;Blkzähler holen
            lda  blktab,y       ;Sekt.nr. lesen,
            sta  mem3           ;ablegen,
            ldx  #<(com1)       ;und Sektor lesen
            ldy  #>(com1)
            jsr  sendcom
            inc  mem1           ;Blkzähler+1
            lda  #0             ;Eintragszähler
            sta  mem2           ;initialisieren
celoop2     ldy  mem2           ;Eintr.z. holen
            cpy  #8             ;Mit 8 vgleichen
            beq  celoop1        ;Ja, also nächsten Block lesen
            lda  enttab,y       ;Nein, also Pos. lesen,
            sta  mem4           ;ablegen,
            ldx  #<(com2)       ;und Pufferzeiger
            ldy  #>(com2)       ;positionieren.
            jsr  sendcom
            inc  mem2           ;Eintr.z+1
            lda  #8             ;Floppy zum
            jsr  talk           ;Senden auf
            lda  #$62           ;Pufferkanal
            jsr  sectlk         ;bewegen.
            jsr  iecin          ;Filetyp holen.
            cmp  #$82           ;u.m. "PRG" vgl.
            bne  celoop2        ;Wenn <>, dann direkt weiter.
            jsr  iecin          ;Sonst Track
            sta  ftrack         ;und Sektor
            jsr  iecin          ;holen und ab-
            sta  fsector        ;legen.
            ldy  #$ff           ;Jetzt Filenamen
celoop3     iny                 ;mit NAME
            jsr  iecin          ;vergleichen
            cmp  name,y
            beq  celoop3
            cpy  #16            ;Wenn 16 Stellen
            beq  ce1            ;dann OK.
            cmp  #160           ;letztes Byte 160 (Endmarkierung)
            bne  celoop2        ;Nein, also weitersuchen.
            lda  name,y         ;Letztes Namens-
            bne  celoop2        ;byte=0? Nein, weitersuchen.
ce1         lda  #8             ;Floppy
            jsr  untalk         ;zurücksetzen.
            lda  mem4           ;Letzte Eintrags-
            clc                 ;position holen,
```

```
                adc  #1             ;1 Addieren
                ldx  #<(com2)       ;und setzen
                ldy  #>(com2)
                jsr  sendcom
                lda  #8             ;Floppy zum
                jsr  listen         ;Empfang auf
                lda  #$62           ;Pufferkanal
                jsr  seclst         ;bereit machen.
                Lda  #18            ;Tracknr.
                jsr  iecout         ;ausgeben.
                lda  dirstrt        ;SektorNr.
                jsr  iecout         ;ausgeben.
                lda  #8             ;Floppy
                jsr  unlist         ;Zurücksetzen
                lda  mem3           ;letzten Dir-
                ldx  #<(com3)       ;block holen,
                ldy  #>(com3)       ;und zurück-
                jmp  sendcom        ;schreiben.
```

Auch hier werden zwei Floppykommandos benötigt. Diesmal jedoch welche mit fehlendem letzten Parameter, der von SEND-COM hinzugefügt wird:

```
com1            .text "u1 2 0 18"   ;Einen DirBlk
                .byte 0             ;lesen.
com2            .text "b-p 2"       ;Auf best.
                .byte 0             ;Byte posit.
```

9. WBLOCKS

Dies ist die zweite, wichtige Routine unseres Programms. Sie füllt die freien Directoryblocks mit den Bytes vom Fileanfang, bis zu dem Punkt, an dem das 'normale' File beginnt. Gleichzeitig werden diese Dirblocks mit Hilfe des Block-Allocate- Befehls als 'belegt' gekennzeichnet:

```
wblocks         lda  #<(text7)      ;"Schreibe Dir-
                ldy  #>(text7)      ;blocks"
                jsr  strout         ;ausgeben.
                ldx  #<(filemem)    ;Zeiger auf
                ldy  #>(filemem)    ;Filemem
                stx  $fd            ;setzen
                sty  $fe
wbloop1         lda  #<(com4)       ;Pufferzeiger
                ldy  #>(com4)       ;auf 0 posi-
                jsr  striec         ;tionieren.
                lda  #8             ;Floppy auf
                jsr  listen         ;Pufferkanal
                lda  #$62           ;Empfangsbereit
                jsr  seclst         ;machen.
                ldy  dsindex        ;Indexzgr. holen
wb4             iny                 ;und erhöhen.
                lda  blktab,y       ;Sektornr. lesen
                bpl  wb1            ;Wenn>0, dann nicht letzter Block
                lda  ftrack         ;Wenn letzter
                jsr  iecout         ;Block, dann
                lda  fsector        ;Starttrack u.
                jsr  iecout         ;-sektor d. norm.
                jmp  wb2            ;Files eintr.
wb1             pha                 ;Sonst Sekt. retten,
                lda  #18            ;Track 18
                jsr  iecout         ;senden,
                pla                 ;Sektor holen
                jsr  iecout         ;und senden.
```

```
wb2         ldy  #0             ;Anschließend
wbloop2     lda  ($fd),y        ;254 Datenbytes
            jsr  iecout         ;senden.
            iny
            cpy  #$fe
            bne  wbloop2
            lda  #8             ;Floppy zurück-
            jsr  unlist         ;setzen.
            lda  $fd            ;Zeiger neu
            clc                 ;positionieren
            adc  #$fe           ;(+254).
            bcc  wb3
            inc  $fe
wb3         sta  $fd
            ldy  dsindex        ; Sektornummer
            lda  blktab,y       ; holen,
            pha                 ;retten,
            ldx  #<(com3)       ;und Sektor
            ldy  #>(com3)       ;schreiben.
            jsr  sendcom
            pla                 ;Sektornummer
            ldx  #<(com8)       ;zurückholen u.
            ldy  #>(com8)       ;Block als 'be-
            jsr  sendcom        ;'legt' kennzeichnen.
            inc  dsindex        ;Index+1
            dec  dirfree        ;freie Dirblocks-1 (dient als Schleifenzähler)
            bne  wbloop1        ;und wiederholen.
            rts
```

Auch hier eine Auflistung der benötigten Floppybefehle. COM3 und COM8 dienen hierbei als Halbbfehle, die von SENDCOM ergänzt werden:

```
com3        .text"u22018"       ;Block auf
            .byte 0             ;T18 schr.
com4        .text "b-p 2 0"     ;Bufferptr.
            .byte 13,0          ;auf 0 setzen
com8        .text "b-a 0 18"    ;Block be-
            .byte 13,0          ;legen.
```

Hiermit ist der Floppykurs nun beendet.

Für diejenigen unter Ihnen, die Geschmack an der Programmierung der Floppy gefunden haben, halten wir für die nächste Ausgabe einen ganz besonderen Leckerbissen parat: ab dann gibt es nämlich einen aufbauenden Floppy-Kurs für Profis, in dem die Floppy direkt programmiert wird.
Übrigens: Sie sollten es nicht auf sich sitzen lassen, daß unser Beispielprogramm mit Einschränkungen arbeitet. Üben Sie sich doch einmal selbst in der Floppyprogrammierung und versuchen Sie das Programm so zu modifizieren, daß auch Files, die kürzer als die vorhandenen freien Dirblocks sind, installiert werden können. Viel Spaß dabei und bei der Programmierung der Floppy an sich wünscht Ihnen,

Uli Basters (ub).

Basic- & Assemblerlistings

Magic Disk 10/92

Listing „FK.NAMECHANGE“

```
10 PRINT"{clr}{rvs on}{7x space}DISKNAMEN UND -ID AENDERN{7x space}"
20 PRINT"{cursor unten}{cursor unten}MOECHTEN SIE:"
30 PRINT"-------------": REM 13 x Minuszeichen
40 PRINT"(N)AMEN DER DISKETTE AENDERN"
50 PRINT"(I)D UND FORMATKENNUNG AENDERN"
60 PRINT"(P)ROGRAMM BEENDENQQ"
70 GETX$:IFX$=""THEN70
80 IF X$="N" THEN GOSUB200:RUN
90 IF X$="I" THEN GOSUB400:RUN
100 IF X$="P" THEN END
110 GOTO70
120 :
130 :
200 OPEN1,8,15,"I":OPEN2,8,2,"#"
210 PRINT#1,"U1 2 0 18 0"
220 PRINT#1,"B-P 2 144"
230 NA$=""
240 FOR I=1 TO 16
250 GET#2,A$:NA$=NA$+A$
260 NEXT
270 PRINT"AKTUELLER NAME: ";NA$
280 INPUT"{5x space}NEUER NAME";NA$
285 IF LEN(NA$)>16 THEN NA$=LEFT$(NA$,16)
290 IF LEN(NA$)<16 THEN NA$=NA$+CHR$(160):GOTO290
300 PRINT#1,"B-P 2 144"
310 PRINT#2,NA$;
320 PRINT#1,"U2 2 0 18 0"
330 PRINT#1,"I"
340 CLOSE1:CLOSE2:RETURN
350 :
360 :
400 OPEN1,8,15,"I":OPEN2,8,2,"#"
410 PRINT#1,"U1 2 0 18 0"
420 PRINT#1,"B-P 2 162"
430 ID$=""
440 FOR I=1 TO 16
450 GET#2,A$:ID$=ID$+A$
460 NEXT
470 PRINT"AKTUELLE ID:";ID$
480 INPUT"{5x space}NEUE ID";ID$
490 IF LEN(ID$)>5 THEN ID$=LEFT$(ID$,5)
500 PRINT#1,"B-P 2 162"
510 PRINT#2,ID$;
520 PRINT#1,"U2 2 0 18 0"
530 PRINT#1,"I"
```

```
540 CLOSE1:CLOSE2:RETURN
```

Listing „FK.SCHREIBSCHUTZ“

```
10 PRINT"{clr}{rvs on}{8x space}DISKETTE SCHREIBSCHUETZEN{7x space}rvs off}"
20 GOSUB 200
30 PRINT"{cursor unten}{cursor unten}WOLLEN SIE:"
40 PRINT"-----------": REM 11 x Minuszeichen
45 X=1
50 IF A$="A" THEN PRINT"(D)ISKETTE SICHERN":X=2:GOTO60
55 PRINT"(D)ISKETTE ENTSICHERN"
60 PRINT"(N)EUE DISK BEHANDELN"
70 PRINT"(P)ROGRAMM BEENDEN"
80 GETX$:IFX$=""THEN80
90 IFX$="N" THEN 500
100 IFX$="P"THEN END
110 IFX$="D"THEN ON X GOSUB 300,400:RUN
120 GOTO80
130 :
140 :
200 OPEN1,8,15,"I"
210 PRINT#1,"M-R";CHR$(1);CHR$(1);CHR$(1)
220 GET#1,A$
230 PRINT"DISKETTE IST R";
240 IFA$="A"THENPRINT"NICHT ";
250 PRINT"GESCHUETZT!"
260 CLOSE1:RETURN
270 :
280 :
300 OPEN1,8,15,"I":OPEN2,8,2,"#"
310 PRINT#1,"U1 2 0 18 0"
320 PRINT#1,"M-W";CHR$(1);CHR$(1);CHR$(1);CHR$(65);
330 PRINT#1,"B-P 2 2"
340 PRINT#2,"A";
350 PRINT#1,"U2 2 0 18 0"
360 PRINT#1,"I"
370 CLOSE1:CLOSE2:RETURN
380 :
390 :
400 OPEN1,8,15,"I":OPEN2,8,2,"#"
410 PRINT#1,"U1 2 0 18 0"
430 PRINT#1,"B-P 2 2"
440 PRINT#2,"B";
450 PRINT#1,"U2 2 0 18 0"
460 PRINT#1,"I"
470 CLOSE1:CLOSE2:RETURN
480 :
490 :
500 PRINT"{cursor unten}{cursor unten}NEUE DISK EINLEGEN UND TASTE DRUECKEN!"
510 GETX$:IFX$=""THEN510
520 RUN
```

Magic Disk 11/92

Listing „FK.UNDEL“

```
10 goto1000
90 :
91 :
92 :
97 rem ********************
98 rem * header ausgeben *
```

```
99 rem ********************
100 print"{weiß}{clr}{rvs on}{17x space}UnDelete{15x space}";
110 print" Written and (c) by Uli Basters in 1992 {hellgrün}"
120 return
187 :
188 :
189 :
197 rem ***********************
198 rem * directory auslesen *
199 rem ***********************
200 print"{cursor unten}{cursor unten}{10x space}Gelesene Files:"
205 open1,8,15,"i":open2,8,2,"#"
220 print#1,"u1 2 0 18 0"
230 gosub600:tr=a
240 gosub600:se=a
245 :
250 dn$="":id$=""
260 print#1,"b-p 2 143"
270 fori=0to15:get#2,a$:dn$=dn$+a$:next
280 print#1,"b-p 2 162"
290 fori=1to5:get#2,a$:id$=id$+a$:next
295 :
299 q=0
300 print#1,"u1 2 0";str$(tr);str$(se)
305 a=int(q/8):dt(a)=tr:ds(a)=se
310 gosub600:tr=a
320 gosub600:se=a
330 forj=1to8
335 print"{cursor oben}{25x cursor rechts}"q
340 gosub600:ty(q)=a
350 gosub600:ft(q)=a
360 gosub600:fs(q)=a
370 x$=""
380 fori=1to16:get#2,a$:x$=x$+a$:next
390 na$(q)=x$
400 fori=1to9:get#2,a$:next
410 gosub600:bl(q)=a
420 gosub600:bl(q)=bl(q)+a*256
425 get#2,a$:get#2,a$
430 q=q+1
440 next
450 iftr<>0then300
460 :
470 close1:close2
480 q=q-1
500 ifna$(q)=""thenq=q-1:goto500
510 return
520 :
521 :
522 :
597 rem **************
598 rem * byte lesen *
599 rem **************
600 get#2,a$
610 ifa$=""then a=0:return
620 a=asc(a$):return
780 :
781 :
782 :
997 rem ******************
```

```
998 rem * hauptprogramm *
999 rem *****************
1000 dim ty(144),ft(144),fs(144),bl(144),na$(144),dt(18),ds(18),li(144)
1005 tt$(0)="DEL":tt$(1)="SEQ":tt$(2)="PRG":tt$(3)="USR":tt$(4)="REL"
1010 poke53280,11:poke53281,11:gosub100
1020 print"{<shift>+<c=> off}{gross/klein}{hellgrün}Beispielprogramm zum Floppy-Kurs Teil 6.";
1030 print"Mit ihm koennen Sie versehentlich"
1040 print"durch 'SCRATCH' geloschte Dateien wie-"
1045 print"derherstellen."
1050 print"Zusaetzlich bietet sich die Moeglich-"
1060 print"keit, ein File vor dem Loeschen zu"
1070 print"schuetzen."
1074 print"{hellrot}{cursor unten}  Bitte Diskette einlegen... [Taste]"
1075 geta$:ifa$=""then1075
1080 gosub100:print"{hellgrün}{3x cursor unten}    Directory wird eingelesen...{hellrot}"
1090 gosub200
1095 :
1100 z=0:gosub100
1105 fori=0toq
1110 if(ty(i)and7)=0thenli(z)=i:z=z+1
1120 next
1130 :
1140 ifz>0then1180
1150 print"{3x cursor unten}{hellgrün}  Keine DEL-Files gefunden! [Taste]"
1160 geta$:ifa$=""then1160
1170 run
1175 :
1180 print"{hellgrün}{2x cursor unten}Gefundene Eintraege:";z;"{cursor unten}{hellrot}"
1190 fori=0to(z-1)
1200 printi")",na$(li(i))
1210 next
1220 input"{hellgrün}{cursor unten}Welcher Eintrag ";n
1230 if(n>=z)or(n<0)then1220
1235 :
1240 gosub100:print"{2x cursor unten}{hellgrün}Zu retten: {hellrot}";na$(li(n))
1250 print"{hellgrün}{cursor unten}0 - DEL"
1260 print"1 - SEQ"
1270 print"2 - PRG"
1280 print"3 - USR"
1290 print"4 - REL"
1300 input"{cursor unten}Welchen Filetyp soll ich zuordnen ";t1
1310 if(n<0)or(n>4)then1300
1320 print"File auch schuetzen (J/N) ?"
1330 geta$:ifa$=""then1330
1340 ifa$="j"thent1=t1+64
1350 t1=t1+128:ty(li(n))=t1
1360 :
1370 gosub100:print"{2x cursor unten}Schreibe neuen Eintrag..."
1380 bl=int(li(n)/8):ei=li(n)-bl*8
1390 tr=dt(bl):se=ds(bl)
1400 open1,8,15,"i":open2,8,2,"#"
1410 print#1,"u1 2 0";tr;se
1420 po=2+ei*32
1430 print#1,"b-p 2";po
1440 print#2,chr$(t1);
1450 print#1,"u2 2 0";tr;se
1460 :
1465 z=0
1470 print"{cursor unten}Belege Blocks neu... "
1480 tr=ft(li(n)):se=fs(li(n))
```

```
1490 print#1,"b-a 0";tr;se
1500 print#1,"u1 2 0";tr;se
1505 z=z+1
1510 gosub600:tr=a
1520 gosub600:se=a
1530 iftr<>0then1490
1540 close1:close2
1550 printz;"Blocks gefunden."
1560 print"{2x cursor unten}{hellgrün}{Weitere Files suchen (J/N) ?"
1565 geta$:ifa$=""then1565
1570 ifa$="n"thenrun
1580 goto1100
```

Listing „FK.DIR“

```
10 gosub200
20 end
30 :
31 :
90 rem *****************
91 rem * zeichen lesen *
92 rem *****************
93 :
100 get#1,a$
110 ifa$=""thena=0:return
120 a=asc(a$):return
130 :
131 :
190 rem ***********************
191 rem * directory anzeigen *
192 rem ***********************
193 :
200 print"“";:open1,8,0,"$"
210 fori=1to4:get#1,a$:next
215 :
220 gosub100:bl=a
230 gosub100:bl=bl+256*a
240 print bl;
250 get#1,a$:printa$;
260 ifa$<>""then250
270 print
280 fori=1to2:get#1,a$:next
290 ifst=0then220
300 close1
310 return
```


Listing „FK.IO.S“

```
1       ;*********************************
10      .ba $9000
11      .eq status      = $90
12      .eq setpar      = $ffba
13      .eq setnam      = $ffbd
14      .eq open        = $ffc0
15      .eq close       = $ffc3
16      .eq chkin       = $ffc6
17      .eq clrch       = $ffcc
18      .eq basin       = $ffcf
19      .eq bsout       = $ffd2
```

```
20      .eq load        = $ffd5
21      .eq save        = $ffd8
22      .eq ckout       = $ffc9
24      .eq intout      = $bdcd
25      .eq inkey       = $ffe4
26      .eq listen      = $ffb1
27      .eq unlist      = $ffae
28      .eq talk        = $ffb4
29      .eq untalk      = $ffab
30      .eq iecin       = $ffa5
31      .eq iecout      = $ffa8
32      .eq sectlk      = $ff96
33      .eq seclst      = $ff93
80      ;*********************************
100     readfile stx$fb
110                     sty $fc
120 ;
130                     ldx #$34
140                     ldy #$03
150                     jsr setnam
160                     lda #01
170                     ldx #08
180                     ldy #$60
190                     jsr setpar
200                     jsr open
210 ;
220                     lda #08
230                     jsr talk
240                     lda #$60
250                     jsr sectlk
260 ;
270                     ldy #00
280     loop1           jsr iecin
290                     sta ($fb),y
300 ;
310                     inc $fb
320                     bne l1
330                     inc $fc
340 ;
350     l1              lda $90
360                     beq loop1
370 ;
380                     lda #08
390                     jsr untalk
400                     lda #01
410                     jmp close
420     ;*********************************
430     writefile       stx $fb
440                     sty $fc
450 ;
460                     ldx #$34
470                     ldy #$03
480                     jsr setnam
490                     lda #01
500                     ldx #08
510                     ldy #$61
520                     jsr setpar
530                     jsr open
540 ;
550                     lda #08
```

```
560                     jsr listen
570                     lda #$61
580                     jsr seclst
590 ;
600                     ldy #00
610     loop2           lda ($fb),y
620                     jsr iecout
630 ;
640                     inc $fb
650                     bn el2
660                     inc $fc
670 ;
680     l2              lda $fe
690                     cmp $fc
700                     bne loop2
710                     lda $fd
720                     cmp $fb
730                     bne loop2
740 ;
750                     lda #08
760                     jsr unlist
770                     lda #01
780                     jmp close
790     ;*********************************
800     errchn          lda #00
810                     jsr setnam
820                     lda #01
830                     ldx #08
840                     ldy #$6f
850                     jsr setpar
860                     jsr open
870 ;
880                     lda #08
890                     jsr talk
900                     lda #$6f
910                     jsr sectlk
920 ;
930     ecloop1         jsr iecin
940                     jsr bsout
950                     lda $90
960                     beq ecloop1
970 ;
980                     lda #08
990                     jsr untalk
1000                    lda #01
1010                    jmp close
```

Listing „FK.SHOWBAM.S“

```
6 ;*********************************
10      .ba $9000
11      .eq status      =$90
12      .eq setpar      =$ffba
13      .eq setnam      =$ffbd
14      .eq open        =$ffc0
15      .eq close       =$ffc3
16      .eq chkin       =$ffc6
17      .eq clrch       =$ffcc
18      .eq basin       =$ffcf
19      .eq bsout       =$ffd2
```

```
20      .eq load        =$ffd5
21      .eq save        =$ffd8
22      .eq ckout       =$ffc9
24      .eq intout      =$bdcd
25      .eq inkey       =$ffe4
26      .eq listen      =$ffb1
27      .eq unlist      =$ffae
28      .eq talk        =$ffb4
29      .eq untalk      =$ffab
30      .eq iecin       =$ffa5
31      .eq iecout      =$ffa8
32      .eq sectlk      =$ff96
33      .eq seclst      =$ff93
40 ;
41      .eq filemem     =$5000
42      .eq name        =$0334
80 ;*********************************
90                      .by $0b,$08,$0a,$00,$9e
91                      .tx "2061"
92                      .by 0,0,0
95 ;*********************************
100     main            lda #00
110                     sta 53280
120                     lda #11
130                     sta 53281
140                     lda #<(text1)
150                     ldy #>(text1)
160                     jsr strout
170 ;
180     mloop5          lda #00
190                     jsr setnam
200                     lda #01
210                     ldx #08
220                     ldy #$6f
230                     jsr setpar
240                     jsr open
250 ;
260                     lda #01
270                     ldx #<(bufnam)
280                     ldy #>(bufnam)
290                     jsr setnam
300                     lda #02
310                     ldx #08
320                     ldy #$62
330                     jsr setpar
340                     jsr open
350      ;***
360                     lda #<(com1)
370                     ldy #>(com1)
380                     jsr sendcom
390                     lda #<(com2)
400                     ldy #>(com2)
410                     jsr sendcom
420 ;
430                     lda #08
440                     jsr talk
450                     lda #$62
460                     jsr sectlk
470 ;
480                     lda #34
```

```
490                     sta $03
500                     lda #163
510                     sta mloop3+1
520 ;
530     mloop3          ldx #163
540                     ldy #04
550                     stx $fb
560                     sty $fc
570                     inc mloop3+1
580 ;
590                     jsr iecin
600                     jsr iecin
610                     jsr byteout1
620                     jsr iecin
630                     jsr byteout1
640                     jsr iecin
650                     ldy $03
660                     ldx sectab,y
670                     jsr byteout2
680 ;
690                     dec $03
700                     bpl mloop3
710 ;
720                     lda #08
730                     jsr untalk
740                     lda #02
750                     jsr close
760                     lda #01
770                     jsr close
780 ;
790     mloop4          jsr inkey
800                     beq mloop4
810 ;
820                     cmp#"←"                 ;in Anführungszeichen 1 = Taste oben links
830                     beq end
840                     jmp mloop5
850     end             jmp $e544
860     ;*********************************
870     strout          sta $fb
880                     sty $fc
890                     ldy #00
900     soloop1         lda ($fb),y
910                     beq soend
920                     jsr bsout
930                     iny
940                     bne soloop1
950                     inc $fc
960                     jmp soloop1
970     soend           rts
980     ;*********************************
990     sendcom         sta $fb
1000                    sty $fc
1010 ;
1020                    lda #08
1030                    jsr listen
1040                    lda #$6f
1050                    jsr seclst
1060 ;
1070                    ldy #00
1080    scloop1         lda ($fb),y
```

```
1090                    beq sc1
1100                    jsr iecout
1110                    iny
1120                    bne scloop1
1130                    inc $fc
1140                    jmp scloop1
1150 ;
1160    sc1             lda #08
1170                    jmp unlist
1180    ;*********************************
1190    byteout1        ldx #07
1200 ;
1210    byteout2        sta $02
1220                    ldy #00
1230 ;
1240    boloop1         lda #46
1250                    ror $02
1260                    bcs bo1
1270                    lda #15
1280    bo1             sta ($fb),y
1290 ;
1300                    lda #40
1310                    clc
1320                    adc $fb
1330                    bcc bo2
1340                    inc $fc
1350    bo2             sta $fb
1360 ;
1370                    dex
1380                    bpl boloop1
1390                    rts
1400    ;*********************************
1410    text1           .tx "..."
1420                    .tx "ShowBAM V1.0 - written 1992 by U.Basters"
1430                    .tx "Programm-Beispiel zum Floppy-Kurs Teil 8."
1440                    .tx "   000000000011111111112222222222233333"
1450                    .by 13
1460                    .tx "   12345678901234567890123456789012345"
1470                    .by 13
1480                    .by " ","0","1",13
1490                    .by " ","0","2",13
1500                    .by " ","0","3",13
1510                    .by " ","0","4",13
1520                    .by " ","0","5",13
1530                    .by " ","0","6",13
1540                    .by " ","0","7",13
1550                    .by " ","0","8",13
1560                    .by " ","0","9",13
1570                    .by " ","1","0",13
1580                    .by " ","1","1",13
1590                    .by " ","1","2",13
1600                    .by " ","1","3",13
1610                    .by " ","1","4",13
1620                    .by " ","1","5",13
1630                    .by " ","1","6",13
1640                    .by " ","1","7",13
1650                    .by " ","1","8",13
1660                    .by " ","1","9",13
1670                    .tx " 20{hellblau}{22x space}{gelb}'Key' next disk."       ;sh. Zeile 820
1680                    .tx " 21{hellblau}{24x space}{gelb}'←' End"
```

```
1690                    .by 0
1700    bufnam          .tx"#"
1710    com1            .tx "u1 2 0 18 0"
1720                    .by 13,0
1730    com2            .tx "b-p 2 4"
1740                    .by 13,0
1750    sectab          .by 0,0,0,0,0,1,1,1,1,1
1760                    .by 1,2,2,2,2,2,2,2,4,4
1770                    .by 4,4,4,4,4,4,4,4,4,4
1780                    .by 4,4,4,4,4
```

Magic Disk 03/93

Listing „FK.USEDIR.SRC“

```
;******************************
        *= $0801
        status          = $90
        setpar          = $ffba
        setnam          = $ffbd
        open            = $ffc0
        close           = $ffc3
        chkin           = $ffc6
        clrch           = $ffcc
        basin           = $ffcf
        bsout           = $ffd2
        load            = $ffd5
        save            = $ffd8
        ckout           = $ffc9
        intout          = $bdcd
        inkey           = $ffe4
        listen          = $ffb1
        unlist          = $ffae
        talk            = $ffb4
        untalk          = $ffab
        iecin           = $ffa5
        iecout          = $ffa8
        sectlk          = $ff96
        seclst          = $ff93

        filemem         = $0f00
        numbuff         = $0332
        need            = $0337
        dskfree         = $0338
        allfree         = $033a
        dirfree         = $033c
        dirstrt         = $033d
        ftrack          = $033e
        fsector         = $033f
        dsindex         = $0340
        combuff         = $0341

        mem0            = $02
        mem1            = $03
        mem2            = $04
        mem3            = $05
        mem4            = $06

        ;*******************************
        ;jmp main
        ;*******************************
```

```
                .byte $0b,$08,$0a,$00,$9e
                .text "2061"
                .byte 0,0,0
;*******************************
main            lda #11
                sta 53280
                sta 53281

                lda #<(text1)
                ldy #>(text1)
                jsr strout
                jsr getin
                cpy #1
                bne m3
                lda #"x"
                cmp name
                bne m3
                jmp 64738

m3              jsr rfile
                jsr getneed

                lda #<(text2)
                ldy #>(text2)
                jsr strout
mloop1          jsr inkey
                beq mloop1

                lda #<(text3)
                ldy #>(text3)
                jsr strout

                jsr wdummy
                jsr openio
                jsr getdskf
                jsr getdirf
                jsr closeio
                jsr statout

                lda dirfree
                bne m1
                lda #<(errtxt1)
                ldy #>(errtxt1)
                jmp errout

m1              cmp need
                bcc m2
                lda #<(errtxt2)
                ldy #>(errtxt2)
                jmp errout

m2              ldy allfree+1
                bne ok
                lda allfree+0
                cmp need
                bcs ok
                lda #<(errtxt3)
                ldy #>(errtxt3)
                jmp errout

ok              lda #19
                sec
                sbc dirfree
                tay
```

```
                lda blktab,y
                sta dirstrt
                sty dsindex

                ldx dirfree
                txa
                dex
                asl a
                eor #$ff
                clc
                adc #1
                adc #<(filemem)
                bcc m5
                inx
m5              sta $fd
                txa
                clc
                adc #>(filemem)
                sta $fe
                jsr wfile

                jsr openio
                jsr chgent
                jsr wblocks
                jsr closeio

                lda #<(text4)
                ldy #>(text4)

errout          jsr strout
eo1             jsr inkey
                beq eo1
                jmp main
;*********************************
statout         lda #<(stext1)
                ldy #>(stext1)
                jsr strout
                lda #0
                ldx need
                jsr intout
                lda #<(stext2)
                ldy #>(stext2)
                jsr strout
                lda #0
                ldx dirfree
                jsr intout

                lda #<(stext3)
                ldy #>(stext3)
                jsr strout
                lda dskfree+1
                ldx dskfree+0
                jmp intout
;*********************************
getneed         ldx #0
                stx need

                lda $f9
                ldx $fa
                sec
                sbc #<(filemem)
                bcs rf1
```

```
                dex
rf1             sta mem1
                txa
                sec
                sbc #>(filemem)
                sta mem2

                rol a
                bcc cn1
                inc need
cn1             clc
                adc mem1
                beq cn2
                bcc cn2
                inc need
cn2             inc need

                lda mem2
                clc
                adc need
cn3             sta need
                rts
;*********************************
getdskf         lda #<(com5)
                ldy #>(com5)
                jsr striec

                lda #8
                jsr talk
                lda #$6f
                jsr sectlk

                jsr iecin
                sta dskfree+0
                jsr iecin
                jsr iecin
                sta dskfree+1

                lda #8
                jmp untalk
;*********************************
getdirf         lda #<(com6)
                ldy #>(com6)
                jsr striec

                lda #<(com7)
                ldy #>(com7)
                jsr striec

                lda #8
                jsr talk
                lda #$62
                jsr sectlk

                jsr iecin
                sta dirfree

                ldx dskfree+1
                clc
                adc dskfree+0
                bcc gf1
                inx
gf1             sta allfree+0
                stx allfree+1
```

```
                lda #8
                jmp untalk
;*********************************
i2a             ldy #(48-1)
ialoop1         iny
                sec
                sbc #100
                bcs ialoop1
                adc #100
                sty numbuff+1

                ldy #(48-1)
ialoop2         iny
                sec
                sbc #10
                bcs ialoop2
                adc #10
                sty numbuff+2

                clc
                adc #48
                sta numbuff+3

                lda #32
                sta numbuff+0
                lda #0
                sta numbuff+4
                rts
;*********************************
rfile           ldx #<(filemem)
                ldy #>(filemem)
                stx $f9
                sty $fa

                lda mem0
                ldx #<(name)
                ldy #>(name)
                jsr setnam
                lda #1
                ldx #8
                ldy #0
                jsr setpar
                jsr open
                ldx #1
                jsr chkin

                ldy #0
loop1           jsr basin
                sta ($f9),y

                inc $f9
                bne l1
                inc $fa

l1              da $90
                beq loop1

                jsr clrch
                lda #1
                jmp close
;*********************************
wfile           lda #<(text5)
                ldy #>(text5)
```

```
                jsr strout

                lda mem0
                ldx #<(name)
                ldy #>(name)
                jsr setnam
                lda #1
                ldx #8
                ldy #1
                jsr setpar
                jsr open
                ldx #1
                jsr ckout

                ldy #0
wfloop1         lda ($fd),y
                jsr bsout

x3              jsr inkey
                bne x3

                inc $fd
                bne wf1
                inc $fe

wf1             lda $fe
                cmp $fa
                bne wfloop1
                lda $fd
                cmp $f9
                bne wfloop1

                jsr clrch
                lda #1
                jmp close
;*********************************
wdummy          lda mem0
                ldx #<(name)
                ldy #>(name)
                jsr setnam
                lda #1
                ldx #8
                ldy #1
                jsr setpar
                jsr open
                ldx #1
                jsr ckout
                jsr bsout
                jsr clrch
                lda #1
                jsr close

                jsr opencom
                lda #<(name-2)
                ldy #>(name-2)
                jsr striec
                lda #1
                jmp close
;*********************************
getin           ldy #0
giloop1         jsr basin
                cmp #13
                beq giend
```

```
                sta name,y
                iny
                cpy #16
                bne giloop1
giend           sty mem0
                lda #0
                sta name,y
                 rts
;*********************************
strout          sta $fb
                sty $fc
                ldy #0
soloop1         lda ($fb),y
                bne so1
                rts

so1             jsr bsout
                iny
                bne soloop1
                inc $fc
                jmp soloop1
;*********************************
striec          sta $fb
                sty $fc

                lda #8
                jsr listen
                lda #$6f
                jsr seclst

                ldy #0
siloop1 lda ($fb),y
                bne si1
                lda #8
                jmp unlist

si1             jsr iecout
                iny
                bne siloop1
                inc $fc
                jmp siloop1
;*********************************
opencom         lda #1
                ldx #<(comnam)
                ldy #>(comnam)
                jsr setnam
                lda #1
                ldx #8
                ldy #$6f
                jsr setpar
                jmp open
;*********************************
openio  jsr opencom
                lda #1
                ldx #<(bufnam)
                ldy #>(bufnam)
                jsr setnam
                lda #2
                ldx #8
                ldy #$62
                jsr setpar
```

```
                jmp open
;*********************************
closeio lda #2
                jsr close
                lda #1
                jmp close
;*********************************
chgent  lda #<(text6)
                ldy #>(text6)
                jsr strout

                lda #1
                sta mem1

celoop1         lda #8
                jsr untalk
                ldy mem1
                lda blktab,y
                sta mem3
                ldx #<(com1)
                ldy #>(com1)
                jsr sendcom
                inc mem1
                lda #0
                sta mem2

celoop2         ldy mem2
                cpy #8
                beq celoop1
                lda enttab,y
                sta mem4
                ldx #<(com2)
                ldy #>(com2)
                jsr sendcom
                inc mem2

                lda #8
                jsr talk
                lda #$62
                jsr sectlk

                jsr iecin
                cmp #$82
                bne celoop2
                jsr iecin
                sta ftrack
                jsr iecin
                sta fsector

                ldy #$ff
celoop3         iny
                jsr iecin
                cmp name,y
                beq celoop3

                cpy #16
                beq ce1
                cmp #160
                bne celoop2
                lda name,y
                bne celoop2

ce1             lda #8
```

```
                jsr untalk

                lda mem4
                clc
                adc #1
                ldx #<(com2)
                ldy #>(com2)
                jsr sendcom

                lda #8
                jsr listen
                lda #$62
                jsr seclst

                lda #18
                jsr iecout
                lda dirstrt
                jsr iecout

                lda #8
                jsr unlist

                lda mem3
                ldx #<(com3)
                ldy #>(com3)
                jmp sendcom
;*********************************
wblocks         lda #<(text7)
                ldy #>(text7)
                jsr strout

                ldx #<(filemem)
                ldy #>(filemem)
                stx $fd
                sty $fe

wbloop1         lda #<(com4)
                ldy #>(com4)
                jsr striec

                lda #8
                jsr listen
                lda #$62
                jsr seclst

                ldy dsindex
wb4             iny
                lda blktab,y
                bpl wb1
                lda ftrack
                jsr iecout
                lda fsector
                jsr iecout
                jmp wb2

wb1             pha
                lda #18
                jsr iecout
                pla
                jsr iecout

wb2             ldy #0
wbloop2         lda ($fd),y
                jsr iecout
```

```
                iny
                cpy #$fe
                bne wbloop2

                lda #8
                jsr unlist

                lda $fd
                clc
                adc #$fe
                bcc wb3
                inc $fe
wb3             sta $fd

                ldy dsindex
                lda blktab,y
                pha
                ldx #<(com3)
                ldy #>(com3)
                jsr sendcom
                pla
                ldx #<(com8)
                ldy #>(com8)
                jsr sendcom
                inc dsindex
                dec dirfree
                bne wbloop1
                rts
;*********************************
sendcom         pha
                stx scloop1+1
                sty scloop1+2

                ldy #0
scloop1         lda $c000,y
                beq sc1
                sta combuff,y
                iny
                jmp scloop1

sc1             sty mem0
                pla
                jsr i2a
                ldy mem0
                ldx #0
scloop2         lda numbuff,x
                beq sc2
                sta combuff,y
                inx
                iny
                jmp scloop2

sc2             lda #13
                sta combuff,y
                iny
                lda #0
                sta combuff,y

                lda #<(combuff)
                ldy #>(combuff)
                jmp striec
;*********************************
com1            .text "u1 2 0 18"
```

```
                .byte 0
com2            .text "b-p 2"
                .byte 0
com3            .text "u2 2 0 18"
                .byte 0
com4            .text "b-p 2 0"
                .byte 13,0
com5            .text "m-r"
                .byte 250,2,3,13,0
com6            .text "u1 2 0 18 0"
                .byte 13,0
com7            .text "b-p 2 72"
                .byte 13,0
com8            .text "b-a 0 18"
                .byte 13,0

comnam          .text "i"
bufnam.text "#"
;*********************************
blktab          .byte 0,1,4,7,10,13,16
                .byte 2,5,8,11,14,17
                .byte 3,6,9,12,15,18,$ff

enttab          .byte 2,34,66,98,130,162
                .byte 194,226,0

                .text "s:"
name            .byte 0,0,0,0,0,0,0,0
                .byte 0,0,0,0,0,0,0,0,0
;*********************************
text1           .text "{clr}{gross/klein}{<shift>+<c=> off}{rvs on}{hellgrau}"
                .text "             UseDir V1"
                .text ".0               "
                .text "   written anno 1992 b"
                .text "y Uli Basters     "
                .text "(Programmbeispiel zum "
                .text "Floppy-Kurs Teil8)"
                .byte 13
                .text "Dieses Programm speich"
                .text "ert ein File auf"
                .byte 13
                .text "Diskette, jedoch unter"
                .text " Mitbenutzung der"
                .byte 13
                .text "sonst unzugaenglichen "
                .text "Directoryblocks!"
                .byte 13,13
                .text "Filename: {hellblau}"
                .byte 0

text2           .byte 13,13
                .text "{gelb}{rvs on}Bitte Zieldiskette e"
                .text "inlegen!"
                .byte 13,13,0
text3           .text "{hellgrau}Untersuche Zieldiskett"
                .text "e..."
                .byte 13,13,0

text4           .text "{hellgrün}File installiert!!"
                .byte 0

text5           .byte 13,13
```

```
                .text "{hellgrau}File wird angelegt..."
                .byte 13,0
text6           .text "Suche und aendere Eint"
                .text "rag..."
                .byte 13,0
text7           .text "Schreibe und belege Di"
                .text "rblocks..."
                .byte 13,13,0

stext1          .text "{hellgrün}Benoetigte Blocks: "
                .byte 0
stext2          .byte 13
                .text " Freie Dir-Blocks: "
                .byte 0
stext3          .byte 13
                .text "Freie Disk-Blocks: "
                .byte 0

errtxt1         .byte 13,13
                .text "{hellrot}Kein Dir-Block mehr fr"
                .text "ei!"
                .byte 0
errtxt2         .byte 13,13
                .text "{hellrot}File ist zu kurz!"
                .byte 0
errtxt3         .byte 13,13
                .text "{hellrot}File ist zu lang!"
                .byte 0
;*******************************
```

Die Steuerzeichen für die Bildschirmausgabe sind mit grauen Hintergrund hervorgehoben. Ich habe mich an die Schreibweise der www.c64-wiki.de für Steuerzeichen gehalten.

{clr}	Löscht Bildschirm	SHIFT + CLR/HOME
{rvs on}	Reverse Modus ein	CTRL + 9 oder CTRL + R
{cursor unten}	Cursor nach unten	CRSR oben / unten
{cursor oben}	Cursor nach oben	CRSR oben / unten
{<shift>+<c=> off}	Zeichensatzumschaltung sperren	CTRL + H
{gross/klein}	auf Zeichensatz mit großen und kleinen Buchstaben umschalten	CTRL + N
{hellgrün}	Farbe Hellgrün	COMMODORE + 6
{hellrot}	Farbe Hellrot	COMMODORE + 3
{hellblau}	Farbe Hellblau	COMMODORE + 7
{gelb}	Farbe Gelb	CTRL + 8
{hellgrau}	Farbe Hellgrau (Grau 3)	COMMODORE + 3