

Inhaltsverzeichnis		Seite 1
Danksagung / Hilfreiche Links		Seite 2
Teil 1	Ausgabe 03/93	Seite 3
Kursübersicht		Seite 3
Block-Befehle		Seite 4
Block-Read (B-R)		Seite 4
Block-Write (B-W)		Seite 5
Buffer-Pointer (B-P)		Seite 5
Block-Eexecute (M-E)		Seite 5
Block-Allocate & Block-Free (B-A & B-F)		Seite 6
Memory-Befehle		Seite 6
Memory-Read (M-R)		Seite 6
Memory-Write (M-W)		Seite 6
Memory-Execute (M-E)		Seite 7
USER-Befehl		Seite 7
Teil 2	Ausgabe 04+05/93	Seite 8
Die BAM		Seite 8
Der Dis-Patcher (Programm)		Seite 10
Aufbau der Directory		Seite 10
Laden in Assembler		Seite 11
Speichern in Assembler		Seite 12
Die Floppy – ein eigenständiger Computer		Seite 13
Teil 3	Ausgabe 06/93	Seite 13
Fehlerkanal auslesen		Seite 13
Interne Programmausführung		Seite 14
Teil 4	Ausgabe 07+08/93	Seite 18
Jobcodes		Seite 18
Das Aufzeichnungsverfahren		Seite 18
Teil 5	Ausgabe 09/93	Seite 21
Whole-Load		Seite 21
Whole-Save		Seite 22
Directory Laden ohne Programmverlust		Seite 23
Kopierschutz		Seite 24
Teil 6	Ausgabe 10/93	Seite 28
Floppyspeeder		Seite 28

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.

Hilfreiche Weblinks

ROM Listings

Ein vollständig dokumentiertes ROM-Listing der 1541 (1541-II ist nahezu identisch) ist unter folgenden Adresse zu finden:

<http://www.ffd2.com/fridge/docs/1541dis.html>

<http://unusedino.de/ec64/technical/aay/c1541/ro41main.htm>

Zeropage-Adressen

Ein Übersicht der Zeropage-Adressen ist unter folgenden Link zu finden:

<http://unusedino.de/ec64/technical/aay/c1541/ra41main.htm>

Teil 1 – Magic Disk 03/93

Bevor wir loslegen möchte ich mich kurz vorstellen: Ich heiße Frank Boldewin und besitze bin seit 1985 Besitzer meines C64'ers! Seit etwa zwei Jahren versuche ich die letzten Geheimnisse der Floppy zu ergründen und bin auch heute noch begeistert bei der Sache! Da sie das erforderliche Basiswissen bereits von meinen Kollegen Uli Baster und seinem überaus ausführlichen Floppy-Kurs vermittelt bekommen haben, möchte ich, um lästige Wiederholungen zu vermeiden, mit meinem Kurs ganz andere Schwerpunkte setzen. Wie werden uns in erster Linie auf die Programmierung von Routinen konzentrieren, die allgemein als kompliziert angesehen werden, wie z. B. ein Fast-Loader oder ein Kopierschutz.

Sie brauchen keine Angst zu haben, dass dieser Kurs nur für Profis gemacht ist, die soweit fortgeschritten sind, dass man ihnen kaum noch etwas beibringen kann. Jede Routinen die ich ihnen im Laufe dieses Kurses vorstelle, wird genaustens erklärt, so daß sie gegen Ende des Kurses ganz allein in der Lage sein werden ihren eigenen Fast-Loader zu schreiben.

Floppy Programmierer werden sich schwertun, um in den Besitz geeigneter Lektüre zu gelangen. Verglichen mit dem geradezu üppigen Literatur Angebot, das für den C64 vorzufinden ist, kann man das derzeit für die Floppy vorliegende Angebot nur als mager bezeichnen. Deshalb werde ich in dieser und den nächsten Ausgaben versuchen, Ihnen die Kunst des Floppyprogrammierens von Grund auf zu vermitteln!

Anhand von vielen Beispielen, detaillierten Erklärungen und unter Vermeidung langatmiger theoretischer Beschreibungen dürfte es eigentlich kein Problem sein ans gewünschte Ziel zu gelangen.

Zunächst aber eine kleine Übersicht über den Inhalt des gesamten Kurses.

Inhalt:

- Teil 1: 1.Block Befehle
 2.User Befehle
 3.Memory Befehle
 4.der VIA 6522

- Teil 2: 1.die BAM
 2.Aufbau von Files
 3.Laden+Speichern von Files
 4.Zero-Page Adressen

- Teil 3: 1.Fehlerkanal lesen
 2.Lesen+Schreiben eines Blocks
 3.Jobspeicher
 4.Jobcodes

- Teil 4: 1.die Floppy Zero-Page
 2.SYNC Markierungen
 3.Prüfsummen
 4.Aufbau einer formatierten Disk

- Teil 5: 1.Whole-Load+Save
 2.DIR Routine
 3.Kopierschutz

- Teil 6: 1.Diskcontroller
 2.Buscontroller
 3.Fastload

Soweit zum Kursinhalt. In diesem Teil wollen noch einmal sämtliche Floppy-Befehle zusammenfassen und anhand von Basic Routinen kurz erläutern.

Wahrscheinlich haben sie sich schon gefragt, warum man erst ab Kanal 2 mit dem Block-Befehl arbeiten kann. Dies liegt daran, weil die Kanäle 0+1 für das Laden und Speichern schon verwendet werden. Der Kanal 15 wird benötigt um bestimmte Kommandos auszuführen, wie z.B. format, scratch, init usw.! Das gilt natürlich auch für alle anderen Befehle! Schauen wir uns als nächstes doch mal den BLOCK-WRITE Befehl an.

Der BLOCK-WRITE Befehl (B-W)

Dieser Befehl ist das entsprechende Gegenstück zum B-R. Hierbei wird der Inhalt eines Puffers auf Diskette zurückgeschrieben.

Syntax: `Print# Fn,"bw";Cn;Dn;T;S`

Beispiel: `OPEN1,8,2,"#"
OPEN15,8,15
PRINT#15,"B-W 2 0 18 0"
CLOSE1
CLOSE15`

Man sieht schon das er in der Befehlsfolge fast identisch mit dem B-R Befehl ist. Eine ausführliche Erläuterung der Befehlsfolge erübrigt sich daher.

Der BUFFER-POINTER (B-P)

Nehmen wir einmal an Sie möchten anstatt eines ganzen Blocks, nur ein einzelnes Byte aus der Floppy heraus lesen. Kein Problem, den Abhilfe schafft der B-P Befehl.

Dazu muss man wissen, dass ein jeder Buffer einen bestimmten Pointer besitzt. In diesen Pointer kann man nun eine Zahl zwischen 0 und 255 schreiben. Diese Zahl sagt der Floppy welches Byte sie aus der Floppy holen soll. Natürlich muss die Floppy auch wissen aus welchem Track und Sektor. Zum besseren Verständnis nun wieder ein kleines Beispiel mit der entsprechenden Syntax dazu!

Syntax: `Print# Fn,"bp";Cn;Position`

Beispiel: `OPEN1,8,2,"#0"
OPEN15,8,15
PRINT#15,"B-R 2 0 18 0"
PRINT#15,"B-P 2 2"
GET#1,A$
A=ASC(A$+CHR$(0))
PRINT A
CLOSE1
CLOSE15`

Wie Sie sehen lesen wir den Directory-Block in den Floppy-Puffer und holen uns das zweite Byte dieses Sektors mit dem B-P Befehl. Anschließend holen wir uns durch den 'GET'-Befehl das Byte über den Kommando-Kanal ab. Nun kann das Byte auf dem Bildschirm ausgegeben werden! Dies geschieht mit Hilfe von 'PRINT A'.

Dieses gelesene Byte hat eine besondere Funktion auf die ich später im Kurs noch zu sprechen komme!

Der BLOCK-EXECUTE Befehl (M-E)

Der B-E Befehl hat die selbe Syntax wie der B-R Befehl. Seine zusätzliche Eigenschaft ist aber, daß er den geladenen Block im Puffer der Floppy als Maschinenprogramm ausführt. Es erübrigt sich deshalb die Syntax zu diesem Befehl zu erläutern, da er sich wie der B-R Befehl verhält!

Der Block-Allocate (B-A) und Block-Free (B-F):

Stellen sie sich vor Sie haben ein Programm geschrieben, dass bestimmte Daten verwaltet. Für diese Daten möchten Sie jedoch nicht extra Files anlegen und schreiben die Daten auf einen einzelnen Sektor, mithilfe des Direktzugriffs. Alles schön und gut, aber was passiert, wenn man jetzt noch zusätzlich ein Programm auf die selbe Diskette speichern möchte? Sehr wahrscheinlich werden unsere Daten überschrieben, da sie nicht entsprechend gekennzeichnet sind! Um sie nun zu kennzeichnen muss man also den B-A Befehl verwenden! Wir wollen nun T 23, S 1 kennzeichnen!

Syntax: `Print# Fn," ba" ; Dn; T; S`

Beispiel: `Print# Fn," ba 0231"`

Wollen wir den Sektor wieder freigeben, so benutzen wir den B-F Befehl! Die Syntax zum diesem Befehl ist die selbe wie beim B-A Befehl.

Die MEMORY-BEFEHLE

Als nächstes möchte ich mich nun den MEMORY-Befehlen zuwenden. Diese Befehle haben eine ähnliche Bedeutung wie der 'PEEK' und 'POKE' Befehl in Basic, nur mit dem wesentlichen Unterschied, dass nicht die Bytes im Computerspeicher, sondern die im Floppyspeicher gelesen und beschrieben werden können.

Der MEMORY-READ Befehl (M-R)

Mit diesem Befehl kann jede beliebige Speicherstelle der Floppy ausgelesen werden. Verglichen mit den Block-Befehlen sind die Memory-Befehle etwas einfacher zu handhaben, wie Ihnen das folgen Beispiel gleich zeigen wird.

Syntax: `Print# Fn,"M-R" Chr$(LB) Chr$(HB) Chr$(B)`

Bedeutungen der Abkürzungen: LB=Low-Byte Adresse
HB=Hi-Byte Adresse
B =Anzahl der Bytes

Beispiel: `OPEN 15,8,15`
`PRINT#15,"M-R" CHR$(18) CHR$(0) CHR$(2)`
`GET#15,a$,b$`
`PRINT a$;b$`
`CLOSE1`

Mit diesem Befehl wurden die 'ID' Bytes des letzten Lesevorgangs herausgelesen. Diese stehen in der Zeropage in den Speicherstellen 18 und 19 . Wir sehen schon, dass auch hier die entsprechenden Werte mit 'get' abgeholt werden.

Der MEMORY-WRITE Befehl (M-W)

Der M-W Befehl ist das entsprechenden Gegenstück zum M-R Befehl. Mit ihm kann theoretisch jede beliebige Speicherstelle beschrieben werden. Theoretisch deshalb, weil immer nur der Speicherbereich eines Rechners beschrieben werden kann, in dem sich auch tatsächlich RAM befindet. Wie der Speicher der Floppy im einzelnen aufgebaut ist, wird am Schluss des Kursteils erläutert. Auf die folgende Weise, können sie eine oder mehrere Daten in den Floppy-Puffer schreiben.

Syntax: `Print# Fn," M-W" Chr$(LB) Chr$(HB) Chr$(B) Chr$(Data1)`
`Chr$(Data2)`

Beispiel: `OPEN 15,8,15`
`PRINT#15,"M-R" CHR$(18) CHR$(0) CHR$(2)`

```
GET#15,A$,B$
PRINT A$;B$
CLOSE1
```

Der MEMORY-EXECUTE Befehl (M-E)

Der M-E Befehl entspricht dem SYS-Befehl in Basic. Mit ihm lassen sich Maschinenprogramme im Speicher der Floppy starten. Man benutzt den 'M-E' sehr häufig im Zusammenhang mit dem 'M-W' Befehl, wobei der 'M-W' die Bytefolge einer speziellen Maschinenroutine (Fast-Loader oder Kopierschutz) in den Floppy-Puffer schreibt von wo aus, sie mit einem 'M-E' Befehl gestartet oder initialisiert wird. Die Syntax zum Starten einer Maschinenroutine lautet:

Syntax: `Print# Fn,"M-E" Chr$(LB) Chr$(HB)`

Der USER Befehl

Nachdem wir auch diese Reihe besprochen haben, wollen uns nun dem wohl am häufigsten benutzten Befehlen zu, den 'USER' Befehlen.

Beginnen wir mit dem 'U1'. Mit diesem Befehl lässt sich ein Sektor in einen beliebigen Puffer lesen! Mit dem 'U1' kann man auch den ganze Puffer lesen, was ja bei dem B-R Befehl nicht der Fall war, da er das erste Byte des Sektors nicht lesen konnte. Auch der 'M-R' besitzt diese Fähigkeit! Nun schnell zur Syntax!

Syntax: `Print# Fn" u1";Cn;Dn;T;S`

Beispiel: `Print#15" u1 2 0 18 0"`

mit dem 'U2' Befehl lassen sich Daten auf die Diskette zurückschreiben! Da er die gleiche Syntax besitzt wie der 'U1' möchte ich nicht länger darauf eingehen und mich den U 3-8 zuwenden! Sie entsprechen dem 'M-E'! Der Vorteil ist. Das 'LO+HI' Byte nicht mehr angegeben werden müssen, da jeder User 3-8 eine vorgegebene Startadresse hat, die hier in tabellarischer Form wiedergegeben sind:

Befehl	Startadresse
U3	\$0500
U4	\$0503
U5	\$0506
U6	\$0509
U7	\$050C
U8	\$050F

Der Nachteil der U3-8 Befehle ist, dass lediglich 6 verschiedene Start-Befehle für ihr Programm zur Verfügung stehen.

Es sei deshalb ihnen überlassen, ob sie die 'U3-8' oder lieber den 'M-E' Befehl benutzen (bei dem sie ein Programm an jeder beliebigen Adresse starten können) Der 'U9' Befehl ist in der Lage die Floppystation zwischen dem C64(9+) und dem VC20(9-) Betrieb umzuschalten!

Mit U: wird ein Reset in der Floppy ausgelöst!

Zum Schluss dieses Kursteils möchte ich noch schnell die wichtigsten Speicherinhalte des **VIA6522** angeben:

\$0000	Zero Page
\$0100	Stack
\$0145	Page 1
\$0200	Befehlspeicher
\$0228	Page 2
\$0300	Puffer 0 (Hauptspeicher)
\$0400	Puffer 1 (Dirpuffer 2)
\$0500	Puffer 2 (Benutzerpuffer)
\$0600	Puffer 3 (Dirpuffer 1)
\$0700	Puffer 4 (BAM)
\$0800	nicht benutzt
\$1800	serieller Bus
\$1C00	Laufwerkssteuerung
\$C000	16 KByte ROM Betriebssystem

Okay damit wären wir mit der Einführung am Ende. Im nächsten Kursteil ist die professionelle Programmierung der Floppy dar, bei der auch die Assembler-Freaks unter ihnen auf ihre Kosten kommen werden.

Bis nächsten Monat dann also!

(FB)

Teil 2 – Magic Disk 04+05/93

Willkommen zum zweiten Teil unseres Floppykurses!

Da im ersten Teil dieses Kurses so ziemlich alle Möglichkeiten angesprochen und abgehandelt wurden die bei der Floppyprogrammierung von BASIC aus zu realisieren sind, wollen wir ab diesem Kurs in Assembler weiterprogrammieren, da die Routinen, die wir im Laufe der Zeit entwickeln wollen, in Höchstgeschwindigkeit ablaufen sollen. Damit dürfte bereits gesagt sein, dass dies kein reiner Einsteigerkurs sein kann und soll. Erfahrungen im Umgang mit einem Monitor oder Assembler und Beherrschung der Maschinensprache sind erforderlich. Zur Programmierung der Floppy ist außerdem noch ein ausgereifter Diskmonitor erforderlich. Weil alle vorhandenen Diskmonitore unseren Ansprüchen nicht entsprachen, hatten wir keine andere Wahl, als uns hinzusetzen und einen eigenen zu schreiben. Das Ergebnis unserer Arbeit finden sie auf SEITE-1 dieser Ausgabe. Es trägt die Bezeichnung "DISKPATCHER". Da er bereits in diesem Kursteil eingesetzt wird, sollten sie sich möglichst schnell mit seinen Funktionen vertraut machen.

Die Funktionen werden im Laufe dieses Kurses erklärt. Sollten sie bereits einen Diskmonitor besitzen, mit dem sie gut zurechtkommen, können sie selbstverständlich weiterbenutzen.

Doch nun zum eigentlichen Kurs.

Die BAM

Gleich zu Beginn wollen wir uns mit der "BAM" der 1541 auseinandersetzen!

Die BAM ist der Sektorbelegungsplan der Floppy und ist zu finden in Track 18 und Sektor 0!

Aufbau der BAM:

	Byte	Bedeutung	
\$00	000	Tracknummer für Directorybeginn	
\$01	001	Sektornummer für Directorybeginn	
\$02	002	Formatkennzeichen (A)	
\$03	003	nur für 1571	
\$04	004	Spur 1: Freie Sektoren	
\$05	005	Spur 1: Belegung für Sektoren 0 – 7	Bit=0: Sektor belegt / Bit=1: Sektor frei
\$06	006	Spur 1: Belegung für Sektoren 8 – 16	Bit=0: Sektor belegt / Bit=1: Sektor frei
\$07	007	Spur 1: Belegung für Sektoren 17 – 20 (Sektoren 21 bis 23 nicht vorhanden)	Bit=0: Sektor belegt Bit=1: Sektor frei
\$08	008	Spur 2: Freie Sektoren	
\$09	009	Spur 2: Belegung für Sektoren 0 – 7	Bit=0: Sektor belegt / Bit=1: Sektor frei
\$0A	010	Spur 2: Belegung für Sektoren 8 – 16	Bit=0: Sektor belegt / Bit=1: Sektor frei
\$0B	011	Spur 2: Belegung für Sektoren 17 – 20	Bit=0: Sektor belegt / Bit=1: Sektor frei
\$0C – \$8F	012 – 143	Bedeutung wie Byte 4-7, aber für Spuren 3-35	
\$90 – \$A6	144 – 166	HEADER (Diskname+ID)	
\$A7 – \$FF	167 – 255	von der 1541 unbenutzt	

Zusätzliche Informationen sind zur BAM auf c64-wiki zu finden (<https://www.c64-wiki.de/index.php/BAM>) und in der folgenden Tabelle.

\$90 – \$9F	144 – 159	Diskettenname aufgefüllt mit „SHIFT SPACE“	160 (\$A0)
\$A0 – \$A1	160 – 161	jeweils "Shift Space"	160 (\$A0)
\$A2 – \$A3	162 – 163	Diskettenidentifikation (ID)	
\$A4	164	"Shift Space"	160 (\$A0)
\$A5	165	DOS-Version mit der gearbeitet wird	2 = CBM DOS V2.6
\$A6	166	Kopie von Byte 2	bei 1541: "A" / bei 8050: "C"
\$A7 – \$AA	167 – 170	jeweils "Shift Space"	160 (\$A0)
\$AB – \$B3	171 – 179	Modus	\$00=1541 / \$A0=1571
\$B4 – \$DC	180 – 220	unbenutzt	0
\$DD – \$FF	221 – 255	bei 1541 unbenutzt	(bei 1571 Anzahl der freien Blöcke Spur 36 – 70)

Falls sie sich die BAM einmal näher anschauen wollen, können sie dies problemlos mit dem bereits erwähnten "DISK-PATCHER" tun, der ihnen Einblick in alle Tracks und Sektoren der Diskette gebietet.

Bei der Benutzung vom DISKPATCHER sollten sie immer Bedenken, das willkürliches Ändern der Daten auf einer Diskette die Vernichtung aller Daten nach sich ziehen kann.

Legen Sie sich daher zunächst einmal eine neu formatierte(Experimentier-)Diskette zu, da wir in nächster Zeit, viel mit Direktzugriffen zu tun haben werden. Das dabei auch mal was schiefgehen kann ist auch verständlich!

Der DISKPATCHER

bietet 4 Funktionen!

1. Patchen der Diskette

2. Directory anzeigen
3. Floppykommandos senden (s; n; i; r; etc.)
4. Verlassen des Menüs

Da die Funktionen 2-4 sich praktisch von selbst erklären, wollen wir uns einmal den Patcher selbst zur Brust nehmen! Drückt man die Taste 1, dann gelangt man ins Patchermenü! Über die Tasten 'T' und 'S' lassen sich TRACK und SEKTOR einstellen! Sie werden Hexadezimal dargestellt! Wollen wir uns nun Track 18, Sektor 0 anschauen, müssen wir den Track auf \$12 = #18 einstellen! Drücken Sie nun 'R' um den Sektor zu lesen! Es erscheint der Sektor auf dem Bildschirm, den man nun mit Taste 'E' editieren kann, den Sektor beliebig verändern. Durch die Tasten CRSR-UP + CRSR-DOWN können Sie den Sektor hoch und runter scrollen! Durch drücken der RETURN-Taste, kommen wir aus dem Editmenü wieder raus. Die Funktion 'W' schreibt den gepatchten (veränderten) Sektor auf die Diskette zurück! Durch 'Q' kommen Sie dann wieder ins Hauptmenü!

Der AUFBAU DER DIRECTORY

Als nächstes möchte ich Sie mit dem Aufbau eines Directory-Blocks vertraut machen! In den einzelnen Directory-Blocks befinden sich die Filenamen, die sie beim "Listen" (LOAD"\$",8) der Directory erhalten. Sämtliche Directory-Blocks befinden sich auf TRACK 18. Der erste Directory-Block befindet sich auf TRACK18 + SEKTOR1. Dieser Sektor ist folgendermaßen belegt.

Byte	Bezeichnung
000 – 001	Track+ Sektor für nächsten Directory-Block
002 – 031	Eintrag File 1(siehe unten)
032 – 033	nicht benutzt
	usw.
226 – 255	8.Eintrag

Natürlich möchten Sie nun wissen wie wohl so ein Directoryeintrag aussieht!

Byte	Bedeutung	Bit	Erklärung
000	Byte für Filetyp (PRG; REL; SEQ; USR; DEL)	Bit 0 – 2	0: DEL (gelöschte Datei)
			1: SEQ (Sequentielle Datei)
			2: PRG (PRG-Datei)
			3: USR (USR-Datei)
			4: REL (REL-Datei)
		Bit 3 – 5	unbenutzt
		Bit 6	0=Normal / 1=kein Scratch möglich
		Bit 7	0=File noch offen / 1=File geschlossen
001	Track + Sektor des Startsektors		
003 – 018	Filename		
019 – 020	Start erster Side-Sektor (REL)		
021	Datensatzlänge (REL)		
022 – 025	nicht benutzt		
026 – 027	Zwischenspeicher für DEL		
028 – 029	Blocklänge des Files		

Weiter Informationen sind ebenfalls in c64-wiki (<https://www.c64-wiki.de/index.php/Directory>) zu finden.

DAS LADEN EINES FILES IN ASSEMBLER

Nachdem wir uns nun allerhand Tabellen zu Gemüte geführt haben und wir die grundlegenden Dinge kennen, schauen wir uns nun die ROM-Routinen an, mit denen wir den Direktzugriff auf die Floppy machen wollen!

Nehmen wir einmal an, wir möchten ein Programm laden! Kein Problem, werden Sie jetzt sagen! Man tippt einfach, "LOAD"NAME",8,1" ein und schon wird das Programm geladen! Wir wollen nun aber erkunden, was hinter dem 'LOAD' Befehl steckt und schauen uns die Assemblerroutine dazu an! Sie brauchen hierzu einen SMON oder einen ähnlichen Maschinensprachemonitor. Hier also die Laderoutine:

```

lda #$01          ;(Filenummer)
ldx #$08          ;(Geräteadresse)
ldy #$00          ;(Sekundäradresse)
jsr $fe00         (Fileparameter setzen)
lda #$           (Länge,max $10 Bytes)
ldx #$           (LO-Byte Filename)
ldy #$           (HI-Byte Filename)
jsr $fdf9        (Filnamen setzen)
lda #$00         (Load-Flag/1=Verify-Flag)
ldx #$           (LO-Byte Startadresse)
ldy #$           (HI-Byte Startadresse)
jsr $f49e        (Load)
rts
    
```

Durch JSR\$FE00 werden die Fileparameter gesetzt, damit die Floppy weiß von welchem Gerät Sie laden soll und wohin! Ist die Sekundäradresse nämlich '0', wird das Programm an eine Adresse geladen die Sie angeben! Ist Sie '1', dann werden die Informationen nach \$0801 geladen wo sich der Basicstart befindet, um die Programme mit 'RUN' auszuführen!

Dieses ist natürlich nur dann möglich, wenn Sie eine Anspruchsadresse gepoked haben! Der Filename des zu ladenden Files kann sich irgendwo im Speicher ihres Rechners abgelegt sein. Um das entsprechende File zu laden, muss der Laderoutine im Akku die Länge des Filenames und im X- und Y-Register die Adresse als LO+ HI-Byte angegeben werden haben.

```

lda #$10
ldx #$80
ldy #$30
jsr $fdf9
    
```

Das bedeutet, dass der Filename \$10 Bytes lang ist und bei der Adresse \$3080 zu finden ist.

Durch JSR\$F49E wird dann das Programm geladen! Um zu testen, ob das File 'OK' ist, macht man ganz einfach ein 'verify'. Man muss nur 'lda#\$01' durch 'lda#\$00' ersetzen und schon wird geprüft, ob ein File ok oder defekt ist!

Um dieses herauszufinden können Sie das sogenannte Statusbyte abfragen! Es befindet sich in der Zerpage, bei der Adresse \$90 und hat folgende Belegung:

Bit	Bedeutung
1	Zeitüberschreitung bei IEC-Eingabe
2	Zeitüberschreitung bei IEC-Eingabe
3 – 5	nur für die Datasette
6	Übertragung ist beendet und OK
7	Gerät meldet sich nicht

Soll ein File an eine bestimmte Adresse geladen werden, dann müssen sie folgendes eingeben:

```
lda #$00
ldx #$40
ldy #$10
jsr $f49e
```

Das File wird nun an die Adresse \$1040 geladen, da in 'X' die LO- und in 'Y' die HI-Adresse angegeben werden muß! Die folgende Routine bietet eine andere Möglichkeit ein File zu laden:

```
lda #$01
ldx #$08
ldy #$00
jsr $fe00
lda #$          ;Länge für Filnamen
ldx #$          ;Startadresse für Filnamen
ldy #$          ;setzen
jsr $fdf9       ;SETNAM Parameter setzen
jsr $f34a       ;(open)
ldx #$01       ;(chkin=auf
jsr $f20e       ;Empfang schalten)
jsr $ee13       ;(Startadresse
sta $ae         ;LO+HI
jsr $ee13       ;holen und
sta $af         ;speichern)
ldy #$00       ;(Solange Bytes
m02 jsr $ee13     ;vom Bus
sta ($ae),y    ;holen
inc $ae        ;bis die
bne m01        ;Übertragung
inc $af        ;komplett
m01 bit $90     ;beendet
bvc m02        ;ist)
lda #$01       ;(close
jsr $f291      ;File)
jsr $f333      ;(clrchk)
rts
```

Nachdem das File mit JSR\$F34A geöffnet wurde, wird durch LDX#\$01+ JSR\$F20E die Floppy auf Empfang geschaltet! Danach liest man durch die 'IECIN' Routine JSR\$EE13 die Startadresse ein und beginnt dann das File Byteweise zu Übertragen! Zum Schluss wird das File noch durch LDA#\$01+ JSR\$F291 geschlossen, wobei die '1' das zuerst geöffnete File schließt.

Wenn man also zwei Files öffnet muss man darauf achten, welches File man schließen möchte! Die Routine JSR\$F333 verursacht einen CLRCHK und schaltet die Floppy wieder zurück! Durch JSR\$F49E wird praktisch die Routine ab dem 'open' Befehl ersetzt!

DAS SPEICHERN EINES FILES IN ASSEMBLER

Als nächstes wollen wir uns einmal eine Routine ansehen, die ein File speichert. Sie werden bemerken, daß der Aufbau der Speicherroutine große Ähnlichkeit mit dem der Laderoutine hat.

```
lda #$01
ldx #$08
ldy #$00
jsr $fe00
lda #$          ;Länge für Filnamen
ldx #$          ;Startadresse für Filnamen
ldy #$          ;setzen
jsr $fdf9       ;SETNAM Parameter setzen
ldx #$          ;(LO-Startadresse)
ldy #$          ;(HI-Startadresse)
```

```

stx $fb      ;(zwischen-
sty $fc      ;speichern)
lda # $fb    ;(Pointer zeigt zur Startadresse)
ldx # $      ;(LO-Endadresse)
ldy # $      ;(HI-Endadresse)
jsr $f5dd    (Save)
rts
    
```

Nachdem die Fileparameter und der Name übergeben wurden, wird in X und Y die Startadresse angegeben, um zu wissen ab wo die Routine Speichern soll, und speichern diese in \$FB + \$FC zwischen. Danach wird im Akku ein Zeiger auf die Startadresse gelegt und in X und X wird die Endadresse übergeben. Ist das geschafft wird die Saveroutine durch JSR\$F5DD angesprungen. Achten Sie beim angeben der Endadresse darauf, dass Sie 1 Byte mehr angeben, da sonst das letzte Byte nicht gespeichert wird!

Zum Schluss dieses Kursteiles noch schnell eine Tabelle mit Zero-Page- Adressen unter denen die Fileparameter und der Name gespeichert werden:

Adresse	Bedeutung
\$90	Status-Flag
\$93	Flag für Load/Verify
\$98	Anzahl der offenen Files
\$99	Eingabegerät fuer \$FFCF
\$9A	Eingabegerät fuer \$FFCF
\$AE / \$AF	Zähler Filebytes-Start
\$B7	Länge Filename
\$B8	Aktive Filenummer
\$B9	Sekundäradresse
\$BA	Geräteadresse
\$BB / \$BC	Zeiger auf Filenamen

So, nun haben wir es für heute wieder einmal geschafft. Ich hoffe es hat ihnen Spaß gemacht neue Erkenntnis über die Floppy zu sammeln.

Ich würde mich freuen, Sie auch beim nächsten Mal wieder begrüßen zu dürfen!

Bis dahin, Ihr

Frank Boldewin

Teil 3 – Magic Disk 06/93

Willkommen zur dritten Runde unseres Floppykurses. Nachdem wir im letzten Teil das Status-Flag und seine Belegung angesprochen haben, möchten wir Ihnen diesmal ein Programm vorstellen, dass den Fehlerkanal ausließt.

```

lda # $00    ;Status-Flag
sta $90      ;auf 0 setze
lda # $01    ;Filenummer
ldx # $08    ;Geräteadresse
ldy # $6f    ;Sekundäradresse
jsr $fe00    ;Fileparameterjump
lda # $00    ;Länge des
jsr $fdf9    ;Filenamens=0
    
```

```

                jsr $f34a           ;Open
                lda #$08           ;Geräteadresse
                jsr $ed09          ;auf Senden einstellen
                lda #$6f           ;Sekundäradresse
                jsr $edc7          ;übertragen
m01            jsr $ee13          ;Byte empfangen
                jsr $f1ca          ;ausgeben
                bit $90            ;wenn Bit6=0
                bvc m01           ; dann nächstes Byte
                lda #$08          ; Senden durch
                jsr $edef          ; Untalk beenden
                lda #$01          ; Filenummer auf 1
                jsr $f291          ; und Close
                rts
    
```

Beim Starten dieser Routine, gibt Ihnen die Floppy entweder den entsprechenden Fehler aus oder meldet, dass alles 'ok' ist. Über die Subroutine "JSR \$EE13" wird der die Fehlermeldung Byte für Byte von der Floppy zum Computer übertragen und auf dem Bildschirm ausgegeben. Über Bit 6 kann geprüft werden, wann das Ende der Fehlermeldung erreicht ist. Ist Bit 6=0, so bedeutet dies, dass noch nicht alle Bytes der Fehlermeldung übertragen wurden. Es wird also solange zu " JSR\$EE31" zurückgesprungen, bis die die Floppy mit einem gesetzten Bit 6 das Ende der Übertragung signalisiert.

INTERNE-PROGRAMMAUSFÜHRUNG

In dem folgenden Abschnitt wollen wir uns mit der ganz besonderen Fähigkeit der Floppy befassen, kleinere Routinen "intern" auszuführen. In den vorangegangenen Beispielen haben wir immer nur einfache Floppy-Routinen vom C64 aus gestartet. Doch damit sind die Möglichkeiten der Floppy noch lange nicht erschöpft. Man kann z.B. auch ganze Programme in die Floppy transportieren, die diese dann selbständig ausführt. Dies ist besonders dann wichtig, wenn man einen eigenen Schnellader oder einen Kopierschutz schreiben will. Wie sie vielleicht wissen, handelt es sich bei der VC1541 um ein intelligentes Diskettenlaufwerk mit eigenem Prozessor (6502), Speicherplatz (RAM und ROM) und Betriebssystem (DOS) . Dadurch wird kein Speicherplatz und keine Rechenzeit vom angeschlossenen C64 benötigt. Der C64 braucht der Floppy lediglich Befehle zu übermitteln, die diese dann selbständig ausführt. Im Grunde genommen, ist ihre VC1541 nichts weiter als eine zweiter Computer neben ihrem C64 . Sie haben also nicht nur einen sondern gleich zwei Computer auf ihrem Schreibtisch stehen. Im Normalbetrieb muss die Floppy drei verschiedenen Aufgaben gleichzeitig erledigen. Dazu gehören:

1. Durchführung der Datenübertragung von und zum C64.
2. die Interpretation der Befehle und die Verwaltung von Dateien, der zugeordneten Übertragungskanäle und der Blockbuffer.
3. die hardwaremäßige Ansteuerung der Diskette.

Mit Hilfe einer ausgereiften IRQ-Technik kann die Floppy diese drei Aufgaben praktisch gleichzeitig ausführen.

Nachdem wir bereits in den letzten beiden Kursen auf den Aufbau der Diskette eingegangen sind, wollen wir uns einmal ansehen, wie das DOS seine Aufgaben erledigt. Selbstverständlich darf man beim Schreiben von eigenen Routinen im Floppybuffer auch die vorhandenen Betriebssystemroutinen verwenden. Ganz so einfach wie im C64, geht es bei der Floppy jedoch nicht.

Das Betriebssystem der Floppy kann man in zwei Teile unterteilen. Der erste Teil ist das Hauptprogramm, das in einer Endlosschleife läuft. Von diesem Teil aus wird hauptsächlich der serielle Bus verwaltet. Fast alle Unterprogrammaufrufe werden mit absoluten Sprüngen ausgeführt und müssen somit mit einem JMP-Befehl zurück ins Hauptprogramm abgeschlossen werden. Diese Routinen können nicht in eigenen Programmen verwendet werden.

Der zweite Teil des Betriebssystems, ist daher um so besser für eigene Programme zu verwenden, denn es handelt sich dabei um ein IRQ-Programm, welches auch als " Jobschleife" bezeichnet wird. Dieses Programm übernimmt die Lese- und Schreiboperationen auf und von Diskette. Bei jedem Interrupt werden die Speicherstellen \$0 bis \$5 der (Floppy) Zeropage, die die Schnittstelle, zwischen dem Hauptprogramm herstellen, auf ihre Werte überprüft. Alle Werte, die gleich oder größer \$80 sind werden als Befehle (Jobs) behandelt und ausgeführt. Jede dieser Speicherstellen (Job-Speicher) bezieht sich auf einen bestimmten Speicherbereich. Zusätzlich gehören zu jedem Job-Speicher noch zwei Speicherstellen, die den Track und den Sektor angeben, auf die sich der Job (Befehl) bezieht.

Die nachfolgende Tabelle, verdeutlicht den Zusammenhang zwischen Job(Adresse) Track-Sektorangabe und Speicherbereich.

Job	Track	Sektor	Speicherbereich
\$00	\$06	\$07	\$0300-\$03FF
\$01	\$08	\$09	\$0400-\$04FF
\$02	\$0A	\$0B	\$0500-\$05FF
\$03	\$0C	\$0D	\$0600-\$06FF
\$04	\$0E	\$0F	\$0700-\$07FF
\$05	\$10	\$11	kein RAM

Die nächste Tabelle zeigt die Bedeutung der einzelnen Job-Codes.

Job-Code	Aufgabe
\$80	Sektor lesen
\$90	Sektor schreiben
\$A0	Sektor verifizieren
\$B0	Sektor suchen
\$C0	Kopfanschlag
\$D0	Programm im Puffer ausführen
\$E0	Programm im Puffer ausführen, vorher Laufwerksmotor einschalten und Kopf positionieren

Was man mit den Job-Codes anfangen, wollen wir anhand von einem Beispiel zeigen. Das nachfolgende Floppyprogramm soll Sektor \$00 von Track \$12 in den Buffer 0 laden. FLOPPYCODE:

```

lda #$12          ;Tracknummer
sta $06          ;übergeben)
lda #$00         ;Sektornummer
sta $07         ;übergeben)
lda #$80         ;Jobcode lesen
sta $00         ;in Jobspeicher
EndSignal lda $00 ;Rückmeldung
           bmi EndSignal ;abwarten)
           rts
    
```

Dieses Programm muss natürlich erst in die Floppy transportiert werden, damit es ausgeführt werden kann. Diese Aufgabe erledigt unser nächstes Programm.

```

lda #$01          ;FLOPPY INITIALISIEREN
ldx #$08
ldy #$6f
    
```

```

jsr $fe00          ;(Filparameter übergeben)
lda #$00
jsr $fdf9          ;(Filename=0)
jsr $f34a          ;(Open)
lda #$08
jsr $ed0c          ;(Listen)
lda #$6f
jsr $edb9          (Seclst)
lda #"I"
jsr $eddd          (Init Floppy)
lda #$08
jsr $edfe          (Unlisten)
*****
,
lda #$08           ;FCODE IN FBUFFER
jsr $ed0c          (Listen)
lda #$6f
jsr $edb9          (Seclst)
lda #"M"
jsr $eddd
lda #"-"
jsr $eddd
lda #"W"
jsr $eddd
lda #$00           ;ADRESSE LO
jsr $eddd
lda #$05           ;ADRESSE HI
jsr $eddd
lda #$11           ;$11 Bytes kpiere
jsr $eddd           ;(m-w,nach $0500)
ldy #$00
m01               lda FCODE,y
jsr $eddd
iny
cpy #$11
bne m01           ;(Floppyroutine in def. Puff)
lda #$08
jsr $edfe (Unlisten)
*****
,
lda #$08
jsr $ed0c          ;(Listen)
lda #$6f
jsr $edb9          ;(Seclst)
lda #"M"
jsr $eddd
lda #"-"
jsr $eddd
lda #"E"
jsr $eddd
lda #$00           ;STARTADRESSE LO
jsr $eddd
lda #$05           ;SRARTADRESSE HI
jsr $eddd           ;(m-e,Start $0500)
lda #$08
jsr $edfe          ;(Unlisten)
*****
,
lda #$08
jsr $ed0c          ;(Listen)
lda #$6f
jsr $edb9          ;(Seclst)
lda #"M"

```



```

        jsr $eddd
        lda #"- "
        jsr $eddd
        lda #"R"
        jsr $eddd
        lda #$00           ;BUFFER ADRESSE
        jsr $eddd
        lda #$03           ;BUFFER ADRESSE
        jsr $eddd
        lda #$00           ;$00 = $0100
        jsr $eddd         ;(M-R,nach$0300
        lda #$08
        jsr $edfe         ;(Unlisten)
        lda #$08
        jsr $ed09         ;(Talk)
        lda #$6f
        jsr $edc7         ;(Sectlk)
        ldy #$00
m02     jsr $ee13
        sta $0400,y
        iny
        bne m02           ;(Sektor am Screen ausgeben)
        lda #$08
        jsr $edef         ;(Untalk)
        lda #$01
        jsr $f291         ;(Close)
        rts
FCODE   lda #$12
        sta $06
        lda #$00
        sta $07
        lda #$80
        sta $00
m03     lda $00
        bmi m03
        rts
    
```

Was noch zu beachten wäre, ist dass Sie immer nur \$20 Hex-Bytes mit einem Schlag in die Floppy übertragen können. Ist ihr Floppyprogramm also länger, müssen sie es schrittweise hineinschreiben. Der Buffer für den Programmcode und der Buffer für die Daten, dürfen nie gleich sein, weil der Programmcode sich während der Ausführung selbst überschreiben würde. Die Folge wäre ein Absturz der Floppy. Um die Routine nun noch besser zu verstehen, erkläre ich Ihnen kurz noch einmal wie ich vorgegangen bin.

1. initialisieren der Floppy.
2. in Puffer 2 (\$0500), wird die Floppyroutine zum Lesen eines Blocks geschrieben.
3. Start des Programms in der Floppy.
4. Einlesen des Blocks in P0 (\$0300).

Von dort aus abholen und auf dem Bildschirm ausgeben. Sie werden sich sicherlich gefragt haben warum ich beim M-R \$00 als Anzahl der zu lesenden Bytes angegeben habe. Weil die Angabe null Bytes zu lesen praktisch gesehen eine sinnlose Aufgabe ist, wird der Wert \$00 intern in \$100 umgewandelt. Es werden also \$0100 Bytes aus dem Floppybuffer geladen.

Da mit diesem Beispiel die Grundstein zu Job-Code Programmierung gelegt wurde, dürfte auch der Direktzugriff in Assembler für Sie kein Problem mehr darstellen. Mit diesen Grundlagen müssten Sie eigentlich auch mit den anderen Job-Codes zurecht kommen. Beim Tüfteln wünsche ich Ihnen viel Spass und verabschiede mich bis zum nächsten Mal!

Teil 4 – Magic Disk 07+08/93

Ich heie Sie herzlich Willkommen zu 4. Teil unseres Floppykurses. Beginnen mchte ich mit den wichtigsten Adressen, die die Floppy-Zeropage bietet. Ich werde im folgenden nur einige Zeropageadressen erklren, da es einfach zu aufwendig wre, alle darzustellen. Ich verweise Sie jedoch auf das Buch 'Floppy Intern', in dem die komplette Zeropage beschrieben steht. Dieses Buch ist normalerweise in guten Bchereien zu haben.

Doch hier nun die wichtigsten Adressen:

Adresse: Bedeutung:

Adresse	Bedeutung
\$0000	Jobspeicher fr Puffer 0
\$0001	Jobspeicher fr Puffer 1
\$0002	Jobspeicher fr Puffer 2
\$0003	Jobspeicher fr Puffer 3
\$0004	Jobspeicher fr Puffer 4
\$0005	Jobspeicher fr Puffer 5
\$0006 / \$0007	Track +Sektor fr Befehl in Puffer 0
\$0008 / \$0009	Track +Sektor fr Befehl in Puffer 1
\$000A / \$000B	Track +Sektor fr Befehl in Puffer 2
\$000C / \$000D	Track +Sektor fr Befehl in Puffer 3
\$000E / \$000F	Track +Sektor fr Befehl in Puffer 4
\$0010 / \$0011	Track +Sektor fr Befehl in Puffer 5
\$0012 / \$0013	ID der Disk im ASCII-Format
\$0016 bis \$001A	Daten im aktuellen Blockheader
\$0016	1. Zeichen der ID
\$0017	2. Zeichen der ID
\$0018	Tracknummer des Blocks
\$0019	Sektornummer des Blocks
\$001A	Prfsumme des Blockheaders
\$001C	Flag fr Schreibschutz auf Disk

DAS AUFZEICHNUNGSVERFAHREN

In dem folgenden Abschnitt wollen wir uns damit befassen, wie die Bits von der Floppyhardware auf die Diskette geschrieben und von dort wieder gelesen werden. Nachdem eine Diskette formatiert wurde, ist sie in 35 Tracks unterteilt, die als konzentrische Ringe angeordnet sind. Der uerste Track hat die Nummer 1 und der Innerste die Nummer 35. Zum Ansteuern der einzelnen Tracks hat das Laufwerk einen sog. Steppermotor, mit dem der Schreib- / Lesekopf ber jeden Track positioniert werden kann.

Diese Tracks wiederum enthalten eine bestimmte Anzahl von Sektoren, die von Auen nach innen abnehmen, da auf einen ueren Track mehr Sektoren passen als auf einen Inneren.

Es stellt sich nun die Frage, wie man den Anfang eines Sektors auf einem Track erkennt. Man msste also bestimmte Byte- oder Bitkombinationen bevorzugt erkennen knnen, die als Daten-

Bytes nicht vorkommen können. Mit 8 Bit ist es möglich 256 Bytekombinationen darzustellen, die aber jedoch auch alle Datenbytes sein könnten. Der Schlüssel zur Lösung liegt darin, ein Byte nicht durch 8, sondern für die Diskettenaufzeichnung durch mehr Bits darzustellen. Dieses Verfahren wird als "Group Code Recording" (GCR) bezeichnet.

Jeder Sektor besteht aus einem BLOCK-HEADER und dem dazugehörigen DATENBLOCK. Sowohl der Block-Header als auch der Datenblock besitzen zu Beginn eine SYNC-Markierung. Stößt der Schreib-/ Lesekopf auf eine solche SYNC-Markierung, dann muss sie nur noch feststellen ob es sich um einen Blockheader oder Datenblock handelt.

Unterschieden werden sie durch das Byte das sich gleich hinter Markierung befindet. Den Blockheader erkennt man an einem \$08 und den Datenblock an einem \$07 Byte Danach folgt noch die Prüfsumme die zur Lesefehlerkontrolle dient. Die nachfolgende Tabelle zeigt den Aufbau eines Headers und eines Datenblocks.

Sync	
\$08	H
Prüfsumme	E
aktueller Sektor	A
aktueller Track	D
ID1	E
ID2	R
Lücke	

	D
Sync	A
\$07	T
Prüfsumme	E
aktueller Sektor	N
aktueller Track	B
ID1	L
ID2	O
Lücke	C
	K

Nachdem sie sich nun mit dem grundlegendem Aufbau der Diskette vertraut gemacht haben, möchte ich etwas näher auf die Synchronmarkierungen eingehen. Wie wir schon wissen, bestehen die Syncs aus 5\$ff Bytes. Stellen Sie sich nun vor, man hätte einen Block voll mit \$ff Bytes. Die Floppy könnte die Syncs von den Datenbytes nicht mehr unterscheiden und das Ergebnis wäre ein totales Chaos bei der Datenorganisation. Aus diesem Grund haben sich die Entwickler der Floppystation die GCR-Codierung einfallen lassen. Damit die Zusammenhänge verständlich werden, möchte ich kurz auf die Technik eingehen, die beim Lesen von Bytes geschieht.

Der Diskcontroller besitzt einen Timer der in bestimmten Zeitabständen feststellt, ob ein Magnetisierungswechsel stattgefunden hat. Bei gleichbleibender Richtung wird ein 0-Bit, bei veränderter Richtung ein 1-Bit dargestellt. Wenn also ein Byte von der Diskette gelesen werden soll, so wartet der Diskcontroller eine bestimmte Zeit die zum Lesen von 8- Bit erforderlich ist. Leider kann ein Laufwerk nicht absolut gleichmäßig gedreht werden, deshalb wird es notwendig nach jedem Magnetisierungswechsel den Timer neu einzustellen, um Lesefehler zu vermeiden. Logischerweise darf es also nicht passieren das zu viele \$00 Bytes hintereinander folgen, da

sonst zu lange keine Laufwerkskontrolle mehr durchgeführt wird.

Natürlich sind auch zu viele 1-Bit nicht gestattet, so sonst ein Sync-Signal ausgelöst werden würde. Deshalb müssen die Daten, bevor sie auf Diskette geschrieben werden, GCR-Codiert werden.

Durch diese Technik wird ausgeschlossen, daß zu viele 0-Bit und 1-Bit direkt hintereinander folgen und somit die Schreib- und Leseelektronik stören. Lediglich Sync-Markierungen, also mehr als 81-Bit, werden vom DOS uncodiert auf die Diskette geschrieben. Es gibt also zwei Schreibarten:

1. Schreiben von Syncs

Es werden 5\$ff Bytes hintereinander geschrieben, die der Orientierung dient.

2. Schreiben von Daten

Hier werden die Byte-Inhalte codiert, da sie von den Syncs unterschieden werden müssen. Hier nun die Umrechnungstabelle für die Binär-GCR Umwandlung:

Hexadezimal	Binär	GCR
\$0	0000	01010
\$1	0001	01011
\$2	0010	10010
\$3	0011	10011
\$4	0100	01110
\$5	0101	01111
\$6	0110	10110
\$7	0111	10111
\$8	1000	01001
\$9	1001	11001
\$A	1010	11010
\$B	1011	11011
\$C	1100	01101
\$D	1101	11101
\$E	1110	11110
\$F	1111	10101

Wie sich erkennen lässt, handelt es sich bei der GCR-Codierung um einen 5- Bit Code. Jedes 4-Bit Nibble das umgewandelt wird, wird praktisch zu einem 5-Bit GCR-Nibble, d.h. ein Byte was vorher aus 8-Bit bestand, wird durch die Codierung zu 10-Bit. Beim GCR-Codieren werden deshalb jeweils immer 4 Byte gleichzeitig umgewandelt. Als Ergebnis erhält man dann logischerweise 5 Byte. Durch diese Technik erhält man für den Diskcontroller ungefährliche Werte. Zum Schluss fehlt eigentlich nur noch die Prüfsumme, die ebenfalls zur Erkennung von eventuellen Fehlern bestimmt ist.

Hier nun die Berechnung der Prüfsumme:

Es werden alle Bytes des Programms addiert und zum Ergebnis noch 2 Bytes der Startadresse hinzugezählt. Dieses Ergebnis besteht aus einem Low- und Hibase. Das Lowbyte ist die Prüfsumme, zu der noch der Übertrag im Hibase addiert werden muss. Das Endergebnis muss immer kleiner als 256 sein. Damit sind wir mal wieder am Ende des Kursteils angelangt.

Nachdem wir uns nun hervorragend mit dem Aufbau und der grundlegenden Technik, die sich dahinter verbirgt, bestens auskennen, möchte ich ab dem nächsten Teil mit dem Entwickeln von nützlichen Utilitys beginnen. Bis dahin,

Teil 5 – Magic Disk 09/93

Ich heie Sie herzlich willkommen zum 5. Teil unseres Floppykurses. Da in den letzten 4 Teilen die Routinen etwas zu kurz gekommen sind, mchte ich Ihnen diesmal folgende Routinen vorstellen:

1. Whole-Load + Whole-Save

Diese beiden Files ermglichen es auch den Bereich ab \$ d000 zu laden und zu speichern!

2. Directory

Dieses File beschreibt eine Directoryroutine, die beim Lesen der BAM nicht den Bereich ab \$0800 zerstrt!

3. Read 41+ Write 41

Diese beiden Files beinhalten eine Lese- und Schreiberoutine fr den Track 41!

Diese Files befinden sich auf ihrer Diskette und knnen mit \$CD00 gestartet werden. Beginnen mchte nun ich mit der Whole-Load Routine. Wie Sie als Assemblerprogrammierer sehr wahrlich wissen lsst sich auch der

Bereich: \$A000-\$BFFF
 \$D000-\$FFFF

nutzen, indem man in die Speicherstelle \$01 Werte zwischen \$34 und \$37 poked. Wie aber lsst sich der Bereich abspeichern?

Von Modulen wie z.B. die Action Replay wissen wir, dass dies ohne weiteres funktioniert. Was aber ist wenn Sie zum Beispiel ein Spiel geschrieben haben mit Highscore-Save und Sie nur noch einen Bereich bei \$D000 frei haben?

Die normale Systemroutine zum laden und Speichern von Files funktioniert hier nicht und auch Ihr Modul kann Ihnen bei diesem Problem nicht helfen!

Ich stelle Ihnen nun zuerst eine Routine zum laden eines Files ab \$D000 vor und danach die dazugehrige Speicherroutine.

```

                                ;Filenummer
                                ;Gertenummer
                                ;Sekundradresse
                                ;setzen
                                ;Lnge Filename
                                ;Lo- und
                                ;Hi-Byte Filename
                                ;setzen
                                ;open
                                ;Gert auf
                                ;Empfang schalten
                                ;Startadresse
                                ;des
                                ;Files
                                ;holen
                                ;1 Byte lesen
                                ;IRQ setzen
                                ;Zhler auf null
                                ;kompletten
                                ;Speicher freigeben
m02     Lda #$01
                                ;Filenummer
                                Ldx #$08
                                ;Gertenummer
                                Ldy #$00
                                ;Sekundradresse
                                jsr $fe00
                                ;setzen
                                lda #$
                                ;Lnge Filename
                                ldx #$
                                ;Lo- und
                                ldy #$
                                ;Hi-Byte Filename
                                jsr $fdf9
                                ;setzen
                                jsr $f34a
                                ;open
                                ldx #$01
                                ;Gert auf
                                jsr $f20e
                                ;Empfang schalten
                                jsr $ee13
                                ;Startadresse
                                sta $ae
                                ;des
                                jsr $ee13
                                ;Files
                                sta $af
                                ;holen
                                jsr $ee13
                                ;1 Byte lesen
                                sei
                                ;IRQ setzen
                                ldy #$00
                                ;Zhler auf null
                                ldx #$34
                                ;kompletten
                                stx $01
                                ;Speicher freigeben

```

```

                sta ($ae),y      ;und abspeichern
                ldx #$37         ;Urzustand
                stx $01          ;wieder herstellen
                cli              ;IRQ löschen
                inc $ae          ;Zähler erhöhen
M01            bne $m01         ;schon Null?
                inc $af          ;dann nächster Block
                bit $90          ;Ende des Files schon
                bvc $m02        ;erreicht,sonst weiter
                jsr $f333        ;Empfang aus
                lda #$01         ;Close
                jmp $f291        ;File
    
```

Wie Sie sehen, bestand der Trick darin, dass wir das File byteweise reingeladen und vor dem Poke in den Speicher einfach den Vektor \$01 verändert haben. Somit stand uns der komplette Speicher zur Verfügung. Das dieser Trick auch beim Speichern funktioniert versteht sich von selbst.

Hier nun die Speicherroutine:

```

                lda #$01         ;Filenummer
                ldx #$08         ;Gerätenummer
                ldy #$00         ;Sekundäradresse
                jsr $fe00        ;setzen
                lda #$          ;laenge Filename
                ldx #$          ;Lo- und
                ldy #$          ;Hi-Byte Filename
                jsr $fdf9        ;setze
                lda #$61         ;Kanal 1+$60 für Save in
                sta $b9          ;Sekundäradresse poken
                jsr $f3d5        ;IEC-Bus eröffnen
                lda #$08         ;Geräteadresse
                jsr $ed0c        ;Listen
                lda #$61         ;Sekundäradresse
                jsr $edb9        ;Seclst
                ldx #$          ;Lo- und
                ldy #$          ;Hi-Byte
                stx $ac          ;der
                sty $ad          ;Startadresse
                ldx #$          ;Lo- und
                ldy #$          ;Hi-Byte
                stx $ae          ;der
                sty $af          ;Endadresse
                lda $ac          ;Startadresse
                jsr $eddd        ;ins
                lda $ad          ;File
M01            jsr $eddd        ;schreiben
                sei              ;IRQ setzen
                ldy #$00         ;Zähler=0 und kompletten
                sty $01          ;Speicher freigeben
                lda ($ac),y      ;Byte holen
                ldy #$37         ;Urzustand wieder
                sty $01          ;herstellen
                cli              ;IRQ löschen
                jsr $eddd        ;und Speichern
                jsr $fcdb        ;diese Routinen werden im
                jsr $fcd1        ;Anschluss hieran
                bcc $m01        ;erklärt
                jsr $edfe        ;Unlisten
                lda #$08         ;Geräteadresse
                jsr $ed0c        ;Listen
    
```

```

lda #$e1          ;Sekundäradresse + Bit 7
jsr $edb9        ;Seclst
jmp $edfe        ;Unlisten
    
```

In dieser Routine haben wir nun zwei Unbekannte. Zunächst mal die Kombination:

```

lda #$61
sta $b9
jsr $f3d5
lda #$08
jsr $ed0c
lda #$61
jsr $edb9
    
```

Diese Routine ist eine andere Form der 'Open' Routine und dient lediglich der neuen Erkenntnis! Dann waren im Code noch 2 Systemadressen die ich angesprungen habe.

```

jsr $fcdb
+ jsr $fcd1
    
```

In \$FCDB steht folgende Routine:

```

inc $ac
bne $m01
inc $ad
m01    rts
    
```

Hier werden eigentlich nur die Counter für den Speicherbereich erhöht. In \$FCD1 steht dann:

```

sec
lda $ac
sbc $ae
lda $ad
sbc $af
rts
    
```

Hier werden die noch verbleibenden von den schon geschriebenen Bytes abgezogen. Damit die Routine so kurz wie möglich bleibt, habe ich diese beiden Systemroutinen angesprungen! Das File Whole-Save speichert den Bereich von \$D000-\$E000 ab und kann mit dem File Whole-Load wieder reingeladen werden.

Das File das abgespeichert und geladen werden kann habe ich 'TEST' genannt! Nachdem wir nun über das Laden und Speichern von Files bestens Bescheid wissen, möchte ich Sie nun mit einer neuen nützlichen Routine vertraut machen, der Directory Routine! Vor allem deshalb nützlich, weil diese Routine bei einer längeren Directory nicht in den eventuellen Programmbereich ab \$0800 reinschreibt!

Denn ich denke, jedem ist es schon mal passiert, der gerade ein tolles Programm geschrieben hat und jetzt gerne wissen möchte, ob noch genug Platz auf der Disk ist. Die Directory wird geladen, zufrieden stellt man fest: Es ist ja noch genug Platz frei! Als nun der letzte Blick auf das Programm erfolgt, muss man voller Wut feststellen, dass die Directory, das neue Programm zum Teil überschrieben hat.

C64 Besitzer die von Beginn an ein Modul besessen haben, werden dieses Dilemma nie erlebt haben, da eigentlich alle Module eine Directory Routine benutzen, die nicht in den Programmbereich ab \$0800 schreibt.

Die Routine die ich Ihnen jetzt vorstelle, verhindert nicht nur, dass ein Programm überschrieben wird, sondern ist auch schneller und kürzer als die System Routine!

```

idx #$08          ;Geräteadresse
ldy #$00          ;Sekundäradresse
jsr $fe00         ;setzen
lda #$01          ;1 Byte ab
    
```

```

ldx #$60          ;$a360='$' für
ldy #$a3          ;BAM Zugriff
jsr $fdf9         ;setzen
jsr $f34a         ;open
ldx #$01          ;Eingabegerät
jsr $f20e         ;setzen
jsr $f157         ;Header holen
jsr $f157         ;mit BASIN
m03 jsr $f157      ;Track + Sektor
jsr $f157         ;holen mit BASIN
lda $90           ;Status-Bit
bne m01           ;testen
jsr $f157         ;in A/X 16-Bit
tax              ;Blockzahl
jsr $f157         ;holen und in
jsr $bdcd         ;Dezimal umrechnen
jsr $ab3b         ;Space
m02 jsr $f157      ;1 Byte vom Filename
jsr $f1ca         ;holen und ausgeben
bne m02           ;Schon alle Bytes?
lda #$0d          ;dann ASCII $0d für
jsr $f1ca         ;Return ausgeben
lda $dc01         ;Komplette
cmp #$7f          ;Directory schon
bne m03           ;ausgegeben?
m01 lda #$01      ;dann
jsr $f291         ;Close
jmp $f333         ;aktiven Kanal schließen
    
```

Wie Sie sehen ist diese Routine gar nicht so schwer zu verstehen. Lediglich zwei neue System Routinen habe ich benutzt!

1. JSR \$BDCCD

Diese Routine wandelt eine Hexadezimale Zahl in eine Dezimale Zahl um. Vorher muss man nur in die Register Akku und X Hi und Lo-Byte der Hexzahl eintragen.

2. JSR \$AB3B

Diese Systemroutine gibt ein Leerzeichen auf dem Bildschirm aus, denn zwischen Blockzahl und Filename ist stets eine Leerstelle.

KOPIERSCHUTZ

Das letzte Thema, mit dem wir uns heute befassen werden ist wohl eines der umstrittensten in der C-64 Geschichte. Ich spreche vom altbekannten Kopierschutz! In den Jahren 1985-1988 lieferten sich Softwarehäuser und Raubkopierer ein packendes Duell!

Die Softwarefirmen steckten sehr viel Zeit und Geld in ihre Kopierschütze! Die Raubkopierer entwickelten unterdessen nach jeder neuen Kopierschutzvariante ein neues Kopierprogramm! Hier ein paar Varianten von Kopierschützen:

1. Halftracks (der Kopf wird nur um einen halben Track bewegt)
2. Speedchange (die Bitrate wurde auf eine andere Geschwindigkeit eingestellt)
3. Readerrors (absichtlich erzeugen eines Fehlers)
4. Format 41 (eine Diskette wird statt 35 Tracks auf 41 Tracks formatiert.) Vom Floppy-DOS aus sind dieses alles Fehler, die nicht verarbeitet werden können, folglich beim kopieren nicht berücksichtigt werden.

Die Kopie enthält also den absichtlich erzeugten Fehler nicht mehr. Hier genau ist der Trick eines Schutzes! Denn fragte man diesen Fehler im Spiel ab und er war nicht mehr vorhanden, waren auf einmal entweder keine Sprites mehr zu sehen oder das Spiel hing sich völlig auf!

Der schlaue User kaufte sich deshalb lieber ein Original um es auch 100% ig Fehlerfrei spielen

zu können! Nicht aber die Raubkopierer, die sofort versuchten ein besseres Kopierprogramm zu schreiben, dass auch den neuen Schutz kopierte!

Hiermit war das Programm zwar kopiert und 100% ig lauffähig, nicht aber der Schutz entfernt! Später fingen dann ein paar Freaks an das Programm nicht nur zu kopieren, sondern auch die Schutzabfrage zu übergehen, so dass das Spiel von jedem X-beliebigen Kopierprogramm kopiert werden konnte.

Es gab fortan also zwei Arten von Softwarefirmengegnern:

1. Die Raubkopierer - sie waren noch recht harmlos für die Firmen, da nur wenige so gute Kopierprogramme besaßen.
2. Die Cracker - sie waren die eigentlichen Firmenschädlinge, da sie den Schutz komplett entfernten und jeder es kopieren konnte.

Soviel zur Geschichte! Ich stelle Ihnen jetzt einen Schutz aus dem oben genannten Sortiment vor. Ich spreche vom Format 41 Schutz! Wie schon erwähnt, muss die Diskette vorher mit einem Disketteneditor auf 41 Tracks formatiert werden. Hier für eignet sich zum Beispiel der Disk-Demon! Nachdem nun die Diskette dementsprechend formatiert wurde, müssen wir zunächst mal eine Write-Routine für den Track 41 entwickeln.

Auf der Diskette befindet sich das dazu gehörige Programm, welches auch die Floppyroutine in die Floppy verfrachtet. Da die Routine zum starten eines Programms in der Floppy eigentlich aus den vergangenen Kursteilen bekannt sein müsste, erkläre ich nun lediglich die Floppyroutine:

```

                                lda #$03                ;Puffer ab
                                sta $31                ;$0300
                                jsr $f5e9             ;Parity für Puffer
                                sta $3a                ;berechnen u. speichern
                                jsr $f78f             ;Puffer in GCR umrechnen
                                jsr $f510             ;Headerblock suchen
                                ldx #$09
m01                               bvc m01                ;Byte ready?
                                clv
                                dex                  ;9 Bytes GAP nach Header-
                                bne m01                ;block überlesen
                                lda #$ff              ;Port A (Read/Writehead)
                                sta $1c03             ;auf Ausgang
                                lda $1c0c             ;PCR auf
                                and #$1f              ;Ausgabe
                                ora #$c0              ;umschalten
                                sta $1c0c             ;CB2 lo
                                lda #$ff              ;Sync
                                ldx #$05              ;5x
                                sta $1c01             ;auf die
                                clv                  ;Diskette schreiben
m02                               bvc m02                ;Byte ready?
                                clv
                                dex
                                bne m02
                                ldy #$bb              ;Bytes $01bb bis $01ff
m04                               lda $0100,y         ;=69 GCR Bytes
m03                               bvc m03                ;auf die
                                clv                  ;Diskette
                                sta $1c01             ;schreiben
                                iny
                                bne m04
m06                               lda ($30),y         ;Datenpuffer 256 Bytes
m05                               bvc m05                ;GCR-Code auf die
                                clv                  ;Diskette schreiben

```

```

                sta $1c01
                iny
                bne m06
m07            bvc m07                ;Byte ready?
                lda $1c0c            ;PCR wieder
                ora #$e0            ;auf Eingabe umschalten
                sta $1c0c            ;CB2 hi
                lda #$00            ;Port A (Read/Writehead)
                sta $1c03            ;auf Eingang
                jsr $f5f2            ;GCR in Normalcode wandeln
                lda #$01            ;Ok
                jmp $f969    Meldung!
    
```

Start der Floppy Routine:

```

m08            idx #$09                ;10 Bytes
                lda text,x            ;Text in
                sta $0300,x            ;Writepuffer
                dex                    ;ab $0300
                bpl m08                ;poken
                idx #$29                ;Track 41
                ldy #$00                ;Sektor 0
                stx $0a                ;abspeichern
                sty $0b                ;und
                lda #$e0                ;Programm
                sta $02                ;ab $0500
m09            lda $02                ;starten
                bmi m09                ;und ausführen
                rts                    ;Ende
                .text "protect41 !"
    
```

Sehen wir uns nun zunächst mal den Start der Floppyroutine an.

Hier wird ein Programm durch Jobcode \$E0 ab \$0500 gestartet um einen Text, der nach \$0300 geschoben wurde, auf die Disk zu schreiben (auf Track 41, Sektor 0). Gehen wir nun noch mal die Schritte zum schreiben eines Blocks auf Diskette durch:

- A. Write Puffer angeben (\$31)
- B. Parität berechnen (\$ F5E9)
- C. Normalcode in GCR-Code wandeln (\$F78F)
- D. Headerblock suchen (\$F510)
- E. 9 Bytes GAP nach dem Headerblock überlesen (Warteschleife)
- F. Port A auf Ausgang (\$1C03)
- G. PCR auf Ausgabe umschalten (\$1C0C)
- H. Syncs (\$\$FF) auf Diskette schreiben
- I. durch die GCR Umwandlung wurden aus 256,325 Bytes (siehe Kurs 4), also 69 Bytes mehr im Ausweichpuffer von \$01BB-\$01FF zuerst geschrieben werden.
- J. dann folgen die restlichen 256 Bytes die von \$0300-\$03FF stehen. Die Adresse zum Lesen und Schreiben von Bytes ist (\$1C01) .
- K. PCR auf Eingabe umschalten (\$1C0C)
- L. Port A auf Eingang (\$1C03)
- M. GCRcode in Normalcode wandeln (\$F5F2)
- N. Programm beenden mit (\$F969)

Unsere Installationsroutine wäre damit beendet! Sehen uns nun als nächstes die Leseroutine bzw. die Schutzabfrage an.

```

                lda #$03                ;Puffer
                sta $31                ;ab $0300
    
```

```

m01      jsr $f50a          ;Datenblockanfang suchen
         bvc m01           ;Byte ready?
         clv
m03      lda $1c01         ;Datenbyte holen
         sta ($30),y       ;und 256 mal
         iny               ;in Puffer
         bne m01           ;schreiben
         ldy #$ba
m02      bvc m02           ;Byte ready?
         clv
         lda $1c01         ;Datenbyte holen
         sta $0100,y       ;und 69 mal
         iny               ;nach $01BA - $01FF
         bne m02           ;schreiben
         jsr $f8e0         ;Daten aus GCR berechnen
         ldx #$00          ;Text mit
m04      lda text,x        ;gelesenen
         cmp $0300,x       ;Daten vergleichen
         bne m03           ;nicht gleich?
         lnx               ;sonst
         cpx #$0a         ;noch ein Byte vergleiche
         bne m04           ;bis alle verglichen
         lda #$01          ;ok
         jmp $f969         ;Meldung!
         text."protect41 !"

```

Start der Floppyroutine:

```

         ldx #$29          ;Track 41
         ldy #$00          ;Sektor 0
         stx $0a           ;poken
         sty $0b           ;und
         lda #$e0          ;Programm
         sta $02           ;bei $0500
m05      lda $02           ;starten und
         bmi m05           ;ausführen
         rts               Ende

```

Gehen wir nun wieder die einzelnen Schritte durch:

1. Readpuffer ab \$0300 (\$30+\$31)
2. Datenblockanfang suchen (\$F50A)
3. 256 Bytes nach \$0300 holen und die restlichen Bytes nach \$01BA - \$01FF mit Adresse (\$1C01) werden Bytes von der Diskette abgeholt.
4. GCRcode in Normalcode wandeln (\$F8E0)
5. Abfrage ob richtiger Text im Speicher, wenn nein= Absturz der Floppy wenn ja = ok
Meldung (\$F969)

Bevor sie die Routinen starten, sollten sie die Floppy stets zuerst Reseten und Initialisieren, da sonst unbeabsichtigt falsche Daten gelesen werden könnten. Sicherlich ist die Abfrage in der Leseroutine leicht zu finden und zu übergehen, für den geübten Cracker! Doch ich denke das Grundprinzip eines Kopierschutzes ist erklärt und Ihnen stehen nun die Türen offen einen besseren, schwerer zu analysierenden Kopierschutz zu entwickeln. Ebenfalls haben sie die leicht veränderte Routinen kennengelernt, die das System für die Jobcodes,\$80- lesen und \$90 schreiben, benutzt.

Beim nächsten Mal beschäftigen wir uns dann mit der Speederprogrammierung und schließen damit auch gleichzeitig unsere Floppykursreihe ab!

Bis dahin,

Teil 6 – Magic Disk 10/93

Herzlich willkommen zum letzten Teil unseres Floppykurses. Nach harter Vorarbeit, ist es endlich soweit, sich als letzte Hürde den Floppyspeeder zu setzen. Dies ist ein Programm, welches es dem User ermöglicht Daten mit vielfacher Geschwindigkeit zu laden.

Ich werde dazu zunächst den Vorgang der seriellen Programmierung erklären, was mit Hilfe von bestimmten Registern geschieht. Danach werden wir genaustens auf den "Floppyspeeder" eingehen, der sich als Objectcode auf Seite 1 dieser Magic Disk befindet.

Zunächst die Tabelle über die CIA Zustände.

C64	Bit	Signal	Richtung	1541	Bit
\$DD00	3	ATN	=>	\$1800	7
\$DD00	5	DATA	=>	\$1800	0
\$DD00	7		<=	\$1800	1
\$DD00	4	CLK	=>	\$1800	2
\$DD00	6		<=	\$1800	3

Bei der Übertragung von Bytes wird zunächst die ATN (Attention) Leitung auf High geschaltet. Im C64 müssen zusätzlich noch Bit 0+1 auf 1 und Bit 2 auf 0 stehen. Daraus ergibt sich also der Wert 11=\$0B, der in \$DD00 geschrieben werden muss, um dem C64 mitzuteilen, dass gleich Daten folgen.

Im C64 gibt es nur eine ATN-Leitung die Ein- und Ausgang steuert. In der Floppy hingegen ist Bit 4 für den Ausgang und Bit 7 für den Eingang des ATN Signals verantwortlich. Da Daten von der Floppy zum C64 fließen, müssen wir also Bit 4 (Ausgang) auf High schalten. In Port \$1800 steht demnach der Wert \$10 Dadurch weiß die Floppy, dass gleich Daten herausgeschickt werden.

Wie Sie vielleicht schon bemerkt haben, lassen sich leider keine ganzen Bytes übertragen, sondern jedes einzelne Bit muss mühevoll durch die Leitung geschoben werden. Dass dieser Vorgang lange Ladezeiten beschert, brauche ich wohl nicht näher zu erklären. Um nun die Daten sauber über die Data-Leitung zu bekommen, müssen diese auf die Mikrosekunde genau getaktet werden. Dieser Vorgang geschieht durch die Clock Leitung. Hierdurch erspart man sich komplizierte Zyklenausgleiche. Ohne diese Leitung wäre beim zweiten Bit die Übertragung beendet, da der VIC die Arbeit des C64 regelmäßig für 40 Zyklen unterbricht, um den Bildschirm aufzufrischen. Dadurch würden C64 und 1541 nicht mehr synchron arbeiten und die Chance, die gewünschten Daten zu bekommen, dahin. Um den Bildschirmaufbau zu verhindern, schreibt man einfach eine 0 in \$D011. Danach sperrt man noch schnell den IRQ mit SEI. Durch diesen Trick ist die Clock-Leitung frei geworden und sie kann zusätzlich zur Übertragung genutzt werden. Das Timing zwischen C64 und 1541 bleibt nun uns selbst überlassen.

Gesagt sei noch, dass in der Floppy die Bits 1+3 (Data+ Clock Ausgang) und im C64 die Bits 6+7 (Data+ Clock Eingang) den Datenverkehr regeln. Im folgenden werde ich die C64 und 1541 Routine zur Übertragung eines Bytes erklären.

1541 Routine: (Beispiel Byte \$EA)

```

                                lda #$ea          ;Byte
                                sta $c1          ;merken
m01                             lda $1800      ;auf Attention
                                bpl m01        ;warten
                                lda #$10       ;Data

```

```

m02      sta $1800          ;setzen
         lda $1800          ;auf C64
         bmi m02           ;warten
         lda #$00
         rol $c1
         rol
         rol
         rol $c1
         rol
         rol
         sta $1800          ;Bits 5 und 7
         lda #$00
         rol $c1
         rol
         rol
         rol $c1
         rol
         rol
         sta $1800          ;Bits 4 und 6
         lda #$00
         rol $c1
         rol
         rol
         rol $c1
         rol
         rol
         sta $1800          ;Bits 1 und 3
         lda #$00
         rol $c1
         rol
         rol
         rol $c1
         rol
         rol
         sta $1800          ;Bits 0 und 2 übertragen
         nop                ;durch
         nop                ;6 Taktzyklen
         nop                ;ausgleichen
         lda #$0f           ;ATN
         sta $1800          ;zurücksetzen
         rts                ;Ende
    
```

Wie Sie vielleicht bemerkt haben, muß man diese Routine sehr sorgfältig entwickeln da die Clock Leitung ihrer eigentlichen Aufgabe entmündigt wurde und nun als zusätzliches Übertragungsbit dient.

Auch bei der C64 Routine ist der Zyklenausgleich nötig wie Sie gleich sehen werden.

```

m01      lda #$0b          ;ATN
         sta $dd00          ;setzen
         lda $dd00          ;auf Data Leitung
         bpl m01           ;warten
         lda #$03          ;ATN
         sta $dd00          ;zurücksetzen
         inc $d020          ;mit
         jsr m02           ;30
         nop                ;Takt-
         nop                ;zyklen
         nop                ;aus-
         dec $d020          ;gleichen
    
```

```

lda $dd00
rol
php
rol
rol $08
plp
rol $08          ;Bits 5 und 7
lda $dd00
rol
php
rol
rol $08
plp
rol $08          ;Bits 4 und 6
lda $dd00
rol
php
rol
rol $08
plp
rol $08          ;Bits 1 und 3
lda $dd00
rol
php
rol
rol $08
plp
rol $08          ;Bits 0 und 2 übertragen
lda $08          ;Byte
eor #$ff        ;invertieren
m02            rts          ;Ende

```

Sie werden sich vielleicht wundern, warum zum Schluss der Routine der ganze Wert invertiert wird. Die Ursache liegt in der Hardware! Die Entwickler haben keine Inverter vor die Ein- und Ausgänge geschaltet, so dass jedes vollständig übertragene Bit softwaremäßig invertiert werden muss. Diese Routinen sind eigenständig und können in der 1541 bzw. im C64 mit JSR angesprungen werden. Auf der Diskette befindet sich ein Fastloader, der sich bei genauerem Betrachten in 5 Teile aufteilen lässt.

1. Eine Routine die das Floppyprogramm in die 1541 verfrachtet.
2. Eine Routine die sich das File vom ersten bis zum letzten Byte in den C64 Speicher holt.
3. Die eben beschriebene Routine zur Übertragung eines Bytes (1541-Routine) .
4. Und die eben beschriebene Routine zur Übertragung eines Bytes (C64-Routine) .
5. Eine Routine die sich das File in den 1541 Speicher holt.

Die Routinen 1 und 2 können wir uns getrost sparen, denn solche Routinen wurden schon in früheren Kursteilen entwickelt. Nummer 3 und 4 haben wir oben schon besprochen. Lediglich Routine 5 bedarf einer kurzen Erläuterung.

```

lda #$12          ;Zähler=0?
sta $1c07        ;nächster Stepperschritt
lda #$03          ;aktueller
sta $31          ;Puffer 0 ($0300)
jsr $f50a        ;Datenblockanfang suchen
m01            bvc m01      ;Ready?
                clv          ;Flag löschen
                lda $1c01     ;1 Byte lesen
                sta ($30),y   ;und ab $0300 speichern
                iny          ;Zähler erhöhen
                bne m01      ;Puffer schon voll?

```

```

m02      ldy #$ba          ;Überlaufpuffer benutzen
         bvc m02         ;Ready?
         clv             ;Flag löschen
         lda $1c01       ;1 Byte lesen
         sta $0100,y     ;und ab $01ba speichern
         iny             ;Zähler erhöhen
         bne m02        ;schon letztes Byte?
         jsr $f8e0       ;GCR-Code umwandeln
         rts            ;Ende
    
```

Diese Routine macht eigentlich nichts anderes als der Jobcode \$80, der für das Lesen von Sektoren verantwortlich ist. Ich habe diese Routine deshalb verwendet weil ein Blick hinter die Kulissen sehr Lehrreich sein kann, wie Sie vielleicht bemerkt haben. Doch nun möchte ich noch ein paar Fragen beantworten, die bei dem einen oder anderen noch offen sind. Die im Code verwendete Adresse \$1C07 steht ursprünglich auf \$3A. Dadurch, dass sie auf \$12 gesetzt wurde, wird bewirkt, dass der Steppermotor schneller reagiert, beim Trackwechsel. Dieser Trick ist sehr nützlich bei vollen Disketten. Die Zeropage-Adressen \$30 / \$31 geben den aktuellen Puffer an, mit dem gearbeitet werden soll. Die Port-Adresse \$1C01 liest Bytes von der Diskette. Da diese Daten noch GCR codiert sind, reicht ein Puffer nicht aus und der Bereich \$01BA-\$0200 wird mitbenutzt. Mit der Routine \$F8E0 wird das ganze dann in Normalcode umgewandelt. Nun sind wir am Ende unseres Floppykurses angelangt und ich hoffe es hat Ihnen Spass und Freude bereitet, in die Wunderwelt der Floppyprogrammierung einzutauchen. Es verabschiedet sich,

Frank Boldewin