Teil 1 - Magic Disk 11/93

Wer kennt sie nicht, die tollen Effekte, die die großen Alchimisten der Programmierkunst aus unserem guten alten "Brotkasten" herauszaubern: mehr als 8 Sprites gleichzeitig auf dem Bildschirm, Side- und Topborderroutinen, die die Grenzen des Bildschirmrahmens sprengen, sich wellenförmig bewegende Logos, Grafikseiten, die sich in atemberaubender Geschwindigkeit zusammenfalten, sowie schillernd bunte 256- farbige Bilder. Man könnte diese Liste noch bis ins Unendliche weiterführen und sie würde dennoch die vollständig sein. In der Tat - was manche Programmierer aus den Siliziummolekülen der Hardware des C64 herauskitzeln hat schon so manch Einen zum Staunen gebracht. Hätte Commodore zur Markteinführung des C64 schon gewusst, was für eine Wahnsinnsmaschine sie da gebaut hatten, hätten sich die Entwickler wahrscheinlich selbst an den Kopf gefasst.
All diese Effekte sind nämlich absolut unbeabsichtigt unserer "Dampfmaschine" der Computerwelt eingepflanzt worden, sondern verdanken ihre Existenz einzig und allein der unermüdlichen Ausdauer der Freaks und Coder, die in nächtelanger Herumtüftelei die Software dazu entwickelten, die den C64 weitaus länger leben ließ, als man je geglaubt hätte.
"Rasterinterrupt" heißt das Zauberwort das, einem "Sesam öffne Dich" gleich, dem Programmierer das Tor zur wunderbaren Welt der Computer-Effekte aufstößt.
Und um genau diese Effekte soll sich in diesem Kurs alles drehen. Die nächsten Monate sollen Sie hier erfahren, wie man sie programmiert, und wir werden versuchen Ihnen das Grundwissen zu vermitteln, neue Routinen selbst entwickeln zu können.
Im ersten Teil dieses Kurses wollen wir uns nun zunächst um die Grundlagen kümmern, und den Fragen" Was ist ein Interrupt?", "Wo kommt er her?" und "Was passiert während eines Interrupts?" auf den Grund gehen. Leider müssen wir Sie auch darauf hinweisen, daß zur Programmierung von Interrupts die gute Kenntnis der Maschinensprache sowie des Befehlssatzes des 6510-Prozessors (so wie er im 64 er seinen Dienst tut) Grundbedingung ist.
Desweiteren sollten Sie einen guten Speichermonitor, bzw. Disassembler zur Hand haben, da wir der Einfachheit halber alle Programmbeispiele als direkt ausführbaren Code der Magic Disk beifügen werden. Ein gutes Hilfsmittel dieser Art ist z. B. der, mittlerweile sehr weit verbreitete,"SMON".

1. Was sind Interrupts?

Das Wort "Interrupt" kommt aus dem Englischen und bedeutet "Unterbrechung". Und nichts anderes tut nun ein Interrupt: er unterbricht den Prozessor bei seiner momentanen Arbeit. Das tut er natürlich nicht einfach aus heiterem Himmel. Vielmehr hat der Programmierer die Möglichkeit bestimmte Bedingungen anzugeben, die einen Interrupt auslösen sollen. Auf diese Weise kann man ganz genau den Zeitpunkt bestimmen, zu dem ein Interrupt auftreten soll. Nun wird der Prozessor jedoch nicht nur einfach angehalten, wenn er eine Interruptanforderung bekommt. Da das Ganze ja auch einen Nutzen haben soll, kann natürlich ein ganz bestimmtes Programm von ihm abgearbeitet werden, das auf das Interruptereignis reagieren soll.
Bevor wir uns jedoch in diese Dinge stürzen, wollen wir erst einmal klären, welche Interrupts der 6510 versteht. Er kennt insgesamt vier Unterbrechungstypen, die von ihm jeweils unterschiedlich behandelt werden. Die Unterscheidung wird ihm schon durch seinen hardwaremäßigen Aufbau ermöglicht. Er verfügt nämlich über drei Eingangsleitungen, die die entsprechenden Interrupts bezeichnen (der vierte Interrupt ist ein besonderer, den wir weiter unten besprechen werden). Die Chips um den Prozessor herum sind nun mit diesen Interrupt-Leitungen verbunden, und können ihm so das

Eintreten eines Unterbrechungsereignisses mitteilen. Sie sehen also, daß Interrupts hardwaremäßig ausgelöst werden. Um nun eine Unterbrechung zu einem bestimmten Zeitpunkt auftreten lassen zu können, sollte man sich ebenfalls in der Programmierung der Hardware auskennen.
Und diese wollen wir in diesem Kurs natürlich auch ansprechen.
Kommen wir nun jedoch zu den vier Interrupttypen. Der Erste von ihnen ist wohl Einer der einfachsten. Es handelt sich um den "RESET", der von keinem externen Chip, sondern vielmehr von einem "externen" Menschen ausgelöst wird. Er dient dazu, den Computer wieder in einen Normalzustand zu versetzen. Da der Prozessor selbst nicht merkt, wann er Mist gebaut hat und irgendwo hängt, muß ihm der Benutzer das durch Drücken eines RESET-Tasters mitteilen. Hieraufhin springt er dann automatisch in ein spezielles Programm im Betriebssystem-ROM und stellt den Einschaltzustand des Rechners wieder her. Ein Reset-Taster ist demnach also direkt mit dem Reset-Interrupteingang des Prozessors verbunden.
Der zweite Interrupttyp heißt "NMI" und kann ausschließlich von CIA-B des C64 ausgelöst werden. Dies ist ein spezieller Chip, der die Kommunikation zwischen externen Geräten und 6510 ermöglicht.
Des weiteren ist die 'RESTORE'- Taste direkt mit dem NMI-Eingang des Prozessors verbunden. Drückt man sie, so wird ebenfalls ein NMI ausgelöst. Auch hier springt der 6510 automatisch in eine spezielle Routine des ROMs und führt ein NMI-Programm aus. Es prüft, ob zusätzlich auch noch die "RUN/ STOP"- Taste gedrückt wurde und verzweigt bei positiver Abfrage in die Warmstartroutine des BASIC-Interpreters.
Der dritte Interrupt ist der wohl am häufigsten benutzte. Er heißt "IRQ" und wird von CIA-Seite A, dem Zwilling von CIA-B, sowie dem Videochip (VIC) ausgelöst.
Gerade weil der VIC diesen Interrupt bedient, ist dies derjenige, mit dem wir in diesem Kurs am meisten arbeiten werden.
Der vierte und letzte Interrupt, ist derjenige unter den Vieren, der keine Leitung zum Prozessor hat. Das liegt daran, daß er ein softwaremäßiger Interrupt ist, mit dem sich der Prozessor quasi selbst unterbricht. Er wird ausgelöst, wenn der 6510 die "BRK"-Anweisung ausführen soll. Demnach heißt er auch "BRK"- Interrupt. Er ist eine sehr einfache Unterbrechung, die eigentlich sehr selten verwendet wird, da sie ja viel bequemer durch ein "JMP" oder "JSR" ersetzt werden kann. Dennoch kann er von Nutzen sein, z.B. wenn man einen Debugger programmiert. Ebenso kann über den BRK z.B. in einen Speichermonitor verzweigt werden, der so die Register oder einen bestimmten Speicherbereich zu einem bestimmten Punkt im Programm anzeigen kann. Er unterscheidet sich von den anderen Interrupts lediglich darin, daß er softwaremäßig ausgelöst wird.

2. IRQ und NMI – ein ungleiches Paar

Diese beiden Interrupttypen sind für uns die Interessantesten, da mit Ihnen Unterbrechungsereignisse der Hardware abgefangen werden, die wir ja exzessiv programmieren werden. Außer, daß sie beide verschiedene Quellen haben, unterscheiden Sie sich auch noch in einem weiteren Punkt: während der NMI IMMER ausgelöst wird, wenn ein Unterbrechungsereignis eintritt, kann der IRQ "maskiert", bzw."abgeschaltet" werden. Dies geschieht über den Assemblerbefehl "SEI", mit dem das Interruptflag des Statusregisters gesetzt wird. Ist dieses Flag nun gesetzt, und es tritt ein IRQ-Interruptereignis ein, so ignoriert der Prozessor schlichtweg das Auftreten dieser Unterbrechung. Dies tut er solange, bis er durch den Assemblerbefehl "CLI" die Instruktion erhält, das Interrupt-Flag wieder zu löschen. Erst dann reagiert er wieder auf eintretende Unterbrechungen. Dies ist ein wichtiger Umstand, den wir auch später bei unseren IRQ-Routinen beachten müssen.

3. Interrruptvektoren - Wegweiser für den Prozessor

Was geschieht nun, wenn der 6510 eine der oben genannten Interruptanforderungen erhält? Er soll also seine momentane Arbeit unterbrechen, um in die Interrupt-Routine zu springen. Damit er Nach Beendigung des Interrupts wieder normal fortfahren kann muß er jetzt einige Schritte durchführen:

1. Ist der auftretende Interrupt ein BRK, so wird zunächst das dazugehörige Flag im Statusregister gesetzt.
2. Anschließend werden High- und Low-Byte (in dieser Reihenfolge) des Programmzählers, der die Speicheradresse des nächsten, abzuarbeitenden Befehls beinhaltet, auf den Stack geschoben.
   Dadurch kann der 6510 beim Zurückkehren aus dem Interrupt wieder die ursprüngliche Programmadresse ermitteln.
3. Nun wird das Statusregister auf den Stack geschoben, damit es bei Beendigung des Interrupts wiederhergestellt werden kann, so daß es denselben Inhalt hat, als bei Auftreten der Unterbrechung.
4. Zuletzt holt sich der Prozessor aus den letzten sechs Bytes seines Adressierungsbereiches (Adressen $FFFA-$FFFF) einen von drei Sprungvektoren. Welchen dieser drei Vektoren er auswählt hängt von der Art des Interrupts ab, der ausgeführt werden soll.Hierzu eine Tabelle mit den Vektoren:

Adresse	**Interrupt**	**Sprungadresse**
$FFFA/$FFFB	NMI	$FE43
$FFFC/$FFFD	RESET	$FCE2
$FFFE/$FFFF	IRQ/BRK	$FF48

Da diese Vektoren im ROM liegen, sind sie schon mit bestimmten Adressen vorbelegt, die die Betriebssystem-Routinen zur Interruptverwaltung anspringen. Wir wollen nun einmal einen Blick auf diese Routinen werfen. Da der RESET keine für uns nützliche Unterbrechung darstellt, wollen wir ihn jetzt und im Folgenden weglassen. Kommen wir zunächst zu der Routine für die IRQ/BRK-Interrupts. Wie Sie sehen haben diese beiden Typen denselben Sprungvektor, werden also von der selben Service-Routine bedient. Da diese jedoch auch einen Unterschied zwischen den beiden machen soll hat sie einen speziellen Aufbau. Hier einmal ein ROM-Auszug ab der Adresse $FF48:

```
FF48:   PHA          ;Akku auf Stapel
FF49:   TXA          ;X in Akku und
FF4A:   PHA          ;auf Stapel
FF4B:   TYA          ;Y in Akku und
FF4C:   PHA          ;auf Stapel
FF4D:   TSX          ;Stackpointer nach
FF4E:   LDA $0104,X  ;Statusreg. holen
FF51:   AND #$10     ;BRK-Flag ausmaskieren
FF53:   BEQ $FF58    ;nicht gesetzt also überspringen
FF55:   JMP ($0316)  ;gesetzt, also über Vektor $0316/$0317 springen
FF58:   JMP ($0314)  ;über Vektor $0314/ $0315 springen
```

Dieses kleine Programm rettet zunächst einmal alle drei Register, indem Sie sie einzeln in den Akku holt und von dort auf den Stack schiebt ($FF48-$FF4C).
Dies ist notwendig, da in der Interruptroutine die Register je ebenfalls benutzt werden sollen, und so die ursprünglichen Inhalte beim Verlassen des Interrupts wiederhergestellt werden können (durch umgekehrtes zurück lesen) . Ab Adresse $FF4D wird nun eine Unterscheidung getroffen, ob ein BRK oder IRQ-Interrupt aufgetreten ist. Ist ersteres nämlich der Fall, so muß das BRK-Flag im Statusregister, das bei Auftreten der

Unterbrechung vom Prozessor auf den Stack geschoben wurde, gesetzt sein.
Durch Zugriff auf den Stack, mit dem Stackpointer als Index in X, wird nun das Statusregister in den Akku geholt.
Dies ist übrigens die einzige Methode, mit der das Statusregister als Ganzes abgefragt werden kann. Natürlich geht das immer nur aus einem Interrupt heraus. Das BRK-Flag ist nun im 4. Bit des Statusregisters untergebracht. Durch eine UND-Verknüpfung mit diesem Bit kann es aus dem Statusregisterwert isoliert werden. Ist der Akkuinhalt nun gleich null, so war das BRK-Flag nicht gesetzt und es muß daher ein IRQ vorliegen. In dem Fall wird auf den JMP-Befehl an Adresse $FF58 verzweigt. Im anderen Fall wird der JMP-Befehl an Adresse $FF55 ausgeführt.
Wie Sie sehen springen diese beiden Befehle über einen indirekten Vektor, der beim IRQ in den Adressen $0314/$0315, beim BRK in $0316/$0317 liegt. Da diese Vektoren im RAM liegen, können Sie auch von uns verändert werden. Das ist also auch der Punkt, an dem wir unsere Interrupts "Einklinken" werden. Durch die Belegung dieser Vektoren mit der Adresse unser eigenen Interruptroutine können wir den Prozessor also zu dieser Routine umleiten.
Werfen wir nun noch einen Blick auf die NMI-Routine, die durch den Vektor bei $FFFA/ $FFFB angesprungen wird. Sie ist noch einfacher aufgebaut und besteht lediglich aus zwei Befehlen:

```
FE43:   SEI             ;IRQs sperren
FE44:   JMP ($0318)     ;über Vektor $0318/ $0319 springen
```

Hier wird lediglich der IRQ gesperrt und anschließend über den Vektor in $0318/$0319 gesprungen. Die Akku, Xund Y-Register werden NICHT gerettet. Das muß unsere Interruptroutine selbst tun.
Zusammenfassend kann man also sagen, daß beim Auslösen eines Interrupts jeweils über einen der drei Vektoren, die im Bereich von $0314-$0319 stehen, gesprungen wird. Der angesprungene Vektor ist dabei interruptabhängig. Hier nochmal eine Übersicht der Interruptvektoren im genannten Bereich:

Adresse	**Interrupt**	**Zieladresse**
$0314/$0315	IRQ	$EA31
$0316/$0317	BRK	$FE66
$0318/$0319	NMI	$FF47

Diese Vektoren werden bei einem Reset mit Standardwerten vorbelegt. Hierbei wird dann die jeweilige Standardroutine des Betriebssystems angesprungen, die den entsprechenden Interrupt bearbeiten soll. Möchten wir eigene Interrupts einbinden, so müssen wir lediglich die Zieladresse der Vektoren auf den Anfang unserer Routine verbiegen.

4. Programmbeispiele

Um das Prinzip des Einbindens eines eigenen Interrupts kennenzulernen, wollen wir nun einmal einen eigenen Interrupt programmieren. Es handelt sich dabei um den einfachsten von allen, den BRK-Interrupt. Hier einmal ein Programmlisting, daß Sie als ausführbares Programm auch auf der Rückseite dieser MD unter dem Namen "BRK-DEMO1" finden. Sie müssen es absolut (mit ",8,1") in den Speicher laden, wo es dann ab Adresse $1000 (dez. 4096) abgelegt wird. Gestartet wird es mit "SYS4096". Sie können es sich auch mit Hilfe eines Disassemblers gerne einmal ansehen und verändern:

```
;*** Hauptprogramm
                1000:   LDX #$1C        ;BRK-Vektor auf
                1002:   LDY #$10        ;$101C verbiegen.
                1004:   STX $0316
                1007:   STY $0317
                100A:   BRK             ;BRK auslösen
                100B:   NOP             ;Füll-NOP
                100C:   LDA #$0E        ;Rahmenfarbe auf dez.14
                100E:   STA $D020       ;zurücksetzen
                1011:   LDX #$66        ;Normalen BRK-Vektor
                1013:   LDY #$FE        ;($FE66) wieder
                1015:   STX $0316       ;einstellen
                1018:   STY $0317
                101B:   RTS             ;und ENDE!
;*** Interruptroutine
                101C:   INC $D020       ;Rahmenfarbe hochzählen
                101F:   LDA $DC01       ;Port B lesen
                1022:   CMP #$EF        ;SPACE-Taste gedrückt?
                1024:   BNE LOOP        ;nein, also Schleife
                1026:   PLA             ;ja, also Y-Register
                1027:   TAY
                1028:   PLA             ;X-Register
                1029:   TAX
                102A:   PLA             ;und Akku vom Stapel holen
                102B:   RTI             ;Interrupt beenden.
```

In den Adressen $1000-$100A setzen wir zunächst einmal Low- und High-Byte des BRK-Vektors auf Adresse $101C, wo unsere BRK-Routine beginnt. Direkt danach wird mit Hilfe des BRK-Befehls ein Interrupt ausgelöst, der, wie wir ja mittlerweile wissen, nach Retten der Prozessorregister über den von uns geänderten BRK-Vektor auf die Routine ab $101C springt.
Selbige tut nun nichts anderes, als die Rahmenfarbe des Bildschirms um den Wert 1 hoch zuzählen, und anschließend zu vergleichen, ob die 'SPACE'- Taste gedrückt wurde. Ist dies nicht der Fall, so wird wieder zum Anfang verzweigt, so daß ununterbrochen die Rahmenfarbe erhöht wird, was sich durch ein Farbschillern bemerkbar macht. Wird die 'SPACE'- Taste nun endlich gedrückt, so kommen die folgenden Befehle zum Zuge. Hier holen wir die Prozessorregister, die von der Betriebssystem-Routine in der Reihenfolge Akku, X, Y auf dem Stack abgelegt wurden, wieder umgekehrt zurück. Das abschließende "RTI" beendet den Interrupt.
Diese Anweisung veranlasst den Prozessor dazu, den alten Programmzähler, sowie das Statusregister wieder vom Stapel zu holen, und an die Stelle im Hauptprogramm zurückzuspringen, an der der BRK ausgelöst wurde. Dies ist logischerweise der Befehl direkt nach dem BRK-Kommando.
So sollte man normalerweise denken, jedoch nimmt BRK eine Sonderstellung diesbezüglich ein. Bei IRQs und NMIs wird tatsächlich der Befehl nach dem zuletzt bearbeiten wieder ausgeführt, jedoch wird beim BRK der Offset 2 auf den Programmzähler hinzuaddiert, weshalb nach Beendigung des Interrupts ein Byte hinter den BRK-Befehl verzweigt wird. Demnach dient der NOP-Befehl, der nach dem BRK kommt, lediglich dem Angleichen an die tatsächliche Rücksprungadresse. Er wird nie ausgeführt, da der Prozessor ja an Adresse $100C weiterfährt. Hier nun setzen wir die Rahmenfarbe wieder auf das gewohnte Hellblau und geben dem BRK-Vektor wieder seine Ursprüngliche Adresse zurück. Würden wir das nicht tun, so würde beim nächsten BRK-Befehl, der irgendwo ausgeführt wird, automatisch wieder zu unserem Bildschirmflackern verzweigt werden. Hier jedoch würde der Computer unter Umständen nicht mehr aus dem Interrupt zurückkehren können, weil wir ja nicht wissen, welche Befehle hinter dem auslösenden

BRK standen (wenn es nicht der unseres Programms an Adresse $100A war).
Um bei JEDEM BRK wieder in einen definierten Zustand zu gelangen, müssten wir das unterbrochene Programm gänzlich stoppen und anschließend wieder zur BASIC- Eingabe zurückkehren. Dies ist eine sehr einfache Übung. Wir müssen lediglich den "Müll" den Prozessor und Betriebssystem-BRK-Routine auf dem Stapel abgelegt haben, mit Hilfe von sechs aufeinanderfolgenden "PLA"- Befehlen von dort wieder wegholen, und anschließend einen BASIC-Warmstart durchführen. Noch einfacher und sauberer geht es jedoch, wenn wir den Stapelzeiger gleich nochmal neu initialisieren. Hierbei werden nämlich auch nicht mehr benötigte Daten, die evtl. vor Auslösen des Interrupts auf dem Stack abgelegt wurden, entfernt.
Demnach kann jedes Programm mit folgendem Interrupt abgebrochen werden:

```
LDX #$FF        ;Stapelzeiger per X-Register
TXS             ;zurücksetzen
JMP $E386       ;BASIC-Warmstart anspringen.
```

Es ist ein beliebiges Programm daß immer am Ende eines Interrupts stehen kann. Es bricht zusätzlich auch den Interrupt ab und kehrt zur BASIC-Eingabe zurück.
Folgendes Programmbeispiel nutzt diese Methode. Zuvor holt es sich jedoch die vom Interrupt auf dem Stapel abgelegten Informationen, und gibt Sie in hexadezimaler Schreibweise auf dem Bildschirm aus. Dadurch haben Sie eine ganz simple Debugging-Kontrolle, mit der Sie Fehler in eigenen Programmen abfangen können.
Wird z. B. ein bestimmter Programmteil angesprungen, der einen BRK-Befehl enthält, so wird diese Routine angesprungen, die Ihnen die Registerinhalte, sowie Zustand des Programmzählers, des Statusregisters und des Stapelzeigers zum Zeitpunkt der Unterbrechung auf dem Bildschirm ausgibt. Hier das Listing:

```
;*** Hauptprogramm
          1000:   LDX #$0B        ;BRK-Vektor auf
          1002:   LDY #$10        ;eigene Routine
          1004:   STX $0316       ;bei $100B
          1007:   STY $0317       ;verbiegen
          100A:   BRK             ;Interrupt auslösen
;*** Interruptroutine
          100B:   LDA #$69        ;Adresse Infotext
          100D:   LDY #$10        ;($1069) laden
          100F:   JSR $AB1E       ;Text ausgeben
          1012:   PLA             ;Inhalt Y-Register
          1013:   JSR $103E       ;ausgeben
          1016:   PLA             ;Inhalt X-Register
          1017:   JSR $103E       ;ausgeben
          101A:   PLA;            ;Akkuinhalt
          101B:   JSR $103E       ;ausgeben
          101E:   PLA             ;Statusregister
          101F:   JSR $103E       ;ausgeben
          1022:   PLA             ;PC Low-Byte holen
          1023:   STA $02         ;und zwischenspeichern
          1025:   PLA             ;PC High-Byte holen
          1026:   JSR $103E       ;und ausgeben
          1029:   LDA #$9D        ;'CRSR' left
          102B:   JSR $FFD2       ;ausgeben
          102E:   LDA $02         ;PC Low-Byte holen
          1030:   JSR $103E       ;und ausgeben
          1033:   TSX             ;Stapelzähler nach
          1034:   TXA             ;Akku übertragen
          1035:   JSR $103E       ;und ausgeben
```

```
1038:   LDX #$FF        ;Stapelzähler
103A:   TXS             ;initialiseren
103B:   JMP $E386       ;BASIC-Warmstart
```

Im Bereich von $1000-$1009 setzen wir, wie gewohnt, den BRK-Interruptvektor auf den Beginn unserer Routine, der in diesem Beispiel bei $100B liegt. In $100A wird ein BRK ausgelöst, der die Interruptroutine sofort anspringt. Diese gibt nun, mit Hilfe der Betriebssystemroutine "STROUT" bei $AB1E, einen kleinen Infotext auf dem Bildschirm aus. Hiernach holen wir uns Y und X-Register, sowie Akku und Statusregister vom Stapel (in dieser Reihenfolge), und zwar in dem Zustand, den sie zum Zeitpunkt, als die Unterbrechung eintrat, innehatten. Die Werte werden dabei einzeln mit der Routine bei $103E auf dem Bildschirm ausgegeben. Diese Routine wandelt den Akkuinhalt in eine hexadezimale Zahl um, die auf dem Bildschirm angezeigt wird. Hinter dieser Zahl gibt sie zusätzlich ein Leerzeichen aus, das als optische Trennung zwischen diesem, und dem nächsten Wert dienen soll. Die genaue Beschreibung erspare ich mir hier, da sie ja eigentlich nichts mit unseren Interrupts zu tun hat.
Im Bereich von $1022-$1032 wird nun zunächst das Low-Byte des alten Programmzählers vom Stapel geholt und in der Speicherzelle $02 zwischengespeichert. Hieraufhin wird das High-Byte geladen und auf dem Bildschirm ausgegeben. Abschließend wird das Low-Byte wieder aus $02 herausgeholt und ebenfalls ausgegeben. Zwischen den beiden Zahlen schicken wir noch ein 'CRSR links'- Zeichen auf den Bildschirm, da das Leerzeichen, das durch unsere Ausgaberoutine zwischen High- und Low-Byte steht, nicht erscheinen soll. Die beiden Werte sollen ja zusammenhängend ausgeben werden.
Abschließend wird der Stapelzeiger in den Akku transferiert und ebenfalls ausgegeben. Da wir zu diesem Zeitpunkt ja schon alle Daten, die durch den Interrupt auf den Stapel gelegt wurden, wieder von dort entfernt haben, entspricht der Stapelzeiger genau dem Wert, der zum Zeitpunkt der Unterbrechung vorhanden war.
Abschließend wird der Stapelzeiger wie oben beschrieben zurückgesetzt, und das Programm verzweigt auf den BASIC-Warmstart, womit wir den Interrupt ohne Verwendung von "RTI", unter Korrektur aller ggf. erfolgten Änderungen am Stapel, verlassen hätten.
Versuchen Sie doch jetzt einmal BRKs von anderen Adressen aus (durch "POKE Adresse,0 : SYS Adresse"), oder in eigenen Programmen auszulösen. Sie werden hierbei immer wieder zu unserer kleinen Registeranzeige gelangen, die das System automatisch wieder ins BASIC zurückführt. Benutzen Sie jedoch nach dem erstmaligen Initialiseren des Programms nach Möglichkeit keinen Speichermonitor mehr, da diese Hilfsprogramme nämlich ähnlich arbeiten wie unser Programm, und somit den von uns eingestellten BRK-Vektor auf ihre eigenen Routinen verbiegen.
Experimentieren Sie einmal ein wenig mit den BRK-Unterbrechungen, um sich die Eigenarten von Interrupts im Allgemeinen anzueignen. Im nächsten Kursteil werden wir uns dann um die Ansteuerung der NMI und IRQ-Unterbrechungen kümmern, bevor wir uns dann im dritten Teil endlich den interessantesten Interrupts, den Raster-IRQs nämlich, zuwenden werden.

(ub/ih)

Teil 2 – Magic Disk 12/93

Im zweiten Teil unseres Interruptkurses wollen wir uns um die Programmierung von IRQund NMI-Interrupts kümmern. Hierbei soll es vorrangig um die Auslösung der Beiden durch die beiden CIA-Chips des C64 gehen.

1. Der Betriebssystem-IRQ

Um einen einfachen Anfang zu machen, möchte ich Ihnen zunächst eine sehr simple Methode aufzeigen, mit der Sie einen Timer-IRQ programmieren können. Hierbei machen wir uns zunutze, daß das Betriebssystem selbst schon standardmäßig einen solchen Interrupt über den Timer des CIA-A direkt nach dem Einschalten des Rechners installiert hat. Die Routine die diesen Interrupt bedient, steht bei Adresse $ EA31 und ist vorrangig für das Cursorblinken und die Tastaturabfrage verantwortlich. Wichtig ist, daß der Timer der CIA diesen IRQ auslöst. Hierbei handelt es sich um eine Vorrichtung, mit der frei definierbare Zeitintervalle abgewartet werden können. In Kombination mit einem Interrupt kann so immer nach einer bestimmten Zeitspanne ein Interruptprogramm ausgeführt werden. Die Funktionsweise eines Timers wollen wir etwas später besprechen. Vorläufig genügt es zu wissen, daß der Betriebssystem- IRQ von einem solchen Timer im sechzigstel-Sekunden-Takt ausgelöst wird. Das heißt, daß 60 Mal pro Sekunde das Betriebssystem-IRQ-Programm abgearbeitet wird. Hierbei haben wir nun die Möglichkeit, den Prozessor über den IRQ-Vektor bei $0314/$0315 auf eine eigene Routine springen zu lassen. Dazu muß dieser Vektor lediglich auf die Anfangsadresse unseres eigenen Programms verbogen werden. Hier einmal ein Beispielprogramm:

```
1000:   SEI          ;IRQs sperren
1001:   LDX #$1E     ;IRQ-Vektor bei
1003:   LDY #$10     ;$0314/$0315 auf eigene
1005:   STX $0314    ;Routine bei $101E
1008:   STY $0315    ;verbiegen
100B:   LDA #00      ;Interruptzähler in Adresse
100D:   STA $02      ;$02 auf 0 setzen
100F:   CLI          ;IRQs wieder erlauben
1010:   RTS          ;ENDE

1011:   SEI          ;IRQs sperren
1012:   LDX #$31     ;IRQ-Vektor bei
1014:   LDY #$EA     ;$0314/$0315 wieder
1016:   STX $0314    ;auf normale IRQ-Routine
1019:   STY $0315    ;zurücksetzen.
101C:   CLI          ;IRQs wieder erlauben
101D:   RTS          ;ENDE

101E:   INC $02      ;Interruptzähler +1
1020:   LDA $02      ;Zähler in Akku holen
1022:   CMP #30      ;Zähler=30?
1024:   BNE 102E     ;Nein, also weiter
1026:   LDA #32      ;Ja, also Zeichen in
1028:   STA $0427    ;$0427 löschen
102B:   JMP $ EA31   ;Und SYS-IRQ anspringen

102E:   CMP #60      ;Zähler=60?
1030:   BNE 103B     ;nein, also weiter
1032:   LDA #24      ;ja, also "X"-Zeichen in
1034:   STA $0427    ;$0427 schreiben
1037:   LDA #00      ;Zähler wieder auf
1039:   STA $02      ;Null setzen
103B:   JMP $EA31    ;und SYS-IRQ anspringen
```

Sie finden dieses Programm übrigens auch als ausführbaren Code auf dieser MD unter dem Namen "SYSIRQ-DEMO". Sie müssen es mit "...,8,1" laden und können es sich mit einem Disassembler anschauen. Gestartet wird es mit SYS4096(=$1000).
Was Sie daraufhin sehen, ist ein "X", das in der rechten, oberen Bildschirmecke im Sekundentakt vor sich hin blinkt.

Wollen wir nun klären, wie wir das zustande gebracht haben:
Bei Adresse $1000-$1011 wird die Interruptroutine vorbereitet und der Interruptvektor auf selbige verbogen. Dies geschieht durch Schreiben des Low- und High-Bytes der Startadresse unserer eigenen IRQ-Routine bei $101E in den IRQ-Vektor bei $0314/$0315. Beachten Sie bitte, daß ich vor dieser Initialisierung zunächst einmal alle IRQs mit Hilfe des SEI-Befehls gesperrt habe. Dies muß getan werden, um zu verhindern, daß während des Verbiegens des IRQ-Vektors ein solcher Interrupt auftritt. Hätten wir nämlich gerade erst das Low-Byte dieser Adresse geschrieben, wenn der Interrupt ausgelöst wird, so würde der Prozessor an eine Adresse springen, die aus dem High-Byte des alten und dem Low-Byte des neuen Vektors bestünde. Da dies irgendeine Adresse im Speicher sein kann, würde der Prozessor sich höchstwahrscheinlich zu diesem Zeitpunkt verabschieden, da er nicht unbedingt ein sinnvolles Programm dort vorfindet. Demnach muß also unterbunden werden, daß solch ein unkontrollierter Interrupt auftreten kann, indem der IRQ mittels SEI einfach gesperrt wird.
Bei $100B-$100F setzen wir nun noch die Zeropage-Adresse 2 auf Null. Sie soll der IRQ-Routine später als Interruptzähler dienen. Anschließend werden die IRQs wieder mittels CLI-Befehl erlaubt und das Programm wird beendet.
Durch das Verbiegen des Interruptvektors und dadurch, daß schon ein Timer-IRQ von Betriebssystem installiert wurde, wird unser Programm bei $101E nun 60 Mal pro Sekunde aufgerufen. Die Anzahl dieser Aufrufe sollen nun zunächst mitgezählt werden. Dies geschieht bei Adresse $101E, wo wir die Zähleradresse bei $02 nach jedem Aufruf um 1 erhöhen. Unser "Sekunden-X" soll nun einmal pro Sekunde aufblinken, wobei es eine halbe Sekunde lang sichtbar und eine weitere halbe Sekunde unsichtbar sein soll. Da wir pro Sekunde 60 Aufrufe haben, müssen wir logischerweise nach 30 IRQs das "X"- Zeichen löschen und es nach 60 IRQs wieder setzen. Dies geschieht nun in den folgenden Zeilen. Hier holen wir uns den IRQ-Zählerstand zunächst in den Akku und vergleichen, ob er schon bei 30 ist.
Wenn ja, so wird ein Leerzeichen (Code 32) in die Bildschirmspeicheradresse $0427 geschrieben. Anschließend springt die Routine an Adresse $ EA31. Sie liegt im Betriebssystem-ROM und enthält die ursprüngliche Betriebssystem-IRQ-Routine, die ja weiterhin arbeiten soll.
Ist die 30 nicht erreicht, so wird nach $102E weiter verzweigt, wo wir prüfen, ob der Wert 60 im Zähler enthalten ist. Ist dies der Fall, so wird in die obig genannte Bildschirmspeicheradresse der Bildschirmcode für das "X"(=24) geschrieben. Gleichzeitig wird der Zähler in Speicherstelle 2 wieder auf 0 zurückgesetzt, damit der Blinkvorgang wieder von Neuem abgezählt werden kann. Auch hier wird am Ende auf die Betriebssystem-IRQ-Routine weiter verzweigt.
Ebenso, wenn keiner der beiden Werte im Zähler stand. Dieser nachträgliche Aufruf des System-IRQs hat zwei Vorteile:
zum Einen werden die Systemfunktionen, die von dieser Routine behandelt werden, weiterhin ausgeführt. Das heißt, daß obwohl wir einen eigenen Interrupt laufen haben, der Cursor und die Tastaturabfrage weiterhin aktiv sind. Zum Anderen brauchen wir uns dabei auch nicht um das zurücksetzen der Timerregister (mehr dazu weiter unten) oder das Zurückholen der Prozessorregister (sie wurden ja beim Auftreten des IRQs auf dem Stapel gerettet - siehe Teil1 dieses Kurses) kümmern, da das alles ebenfalls von der System-Routine abgehandelt wird.
Bleiben nun nur noch die Zeilen von $1011bis $101E zu erläutern. Es handelt sich hierbei um ein Programm, mit der wir unseren Interrupt wieder aus dem System entfernen. Es wird hierbei wie bei der Initialisierung des Interrupts vorgegangen. Nach Abschalten der IRQs wird die alte Vektoradresse $EA31 wieder in $0314/$0315 geschrieben. Dadurch werden die IRQs wieder direkt zur System-IRQ-Routine geleitet. Sie werden nun mittels CLI erlaubt und das Programm wird beendet.

2. Die Programmierung der Timer

Einen Interrupt auf die obig genannte Weise in das System "einzuklinken" ist zwar eine ganz angenehme Methode, jedoch mag es vorkommen, daß Sie für spezielle Problemstellungen damit gar nicht auskommen. Um zum Beispiel einen NMI zu programmieren, kommen Sie um die Initialisierung des Timers nicht herum, da das Betriebssystem diesen Interrupt nicht verwendet. Deshalb wollen wir nun einmal anfangen, in die Eingeweide der Hardware des C64 vorzustoßen um die Funktionsweise der CIA-Timer zu ergründen.
Zunächst einmal sollte erwähnt werden, daß die beiden CIA-Bausteine einander gleichen wie ein Ei dem Anderen. Unterschiedlich ist lediglich die Art und Weise, wie sie im C64 genutzt werden.
CIA-A ist hauptsächlich mit der Tastaturabfrage beschäftigt und übernimmt auch die Abfrage der Gameports, wo Joystick, Maus und Paddles angeschlossen werden.
Sie kann die IRQ-Leitung des Prozessors ansprechen, weswegen sie zur Erzeugung solcher Interrupts herangezogen wird.
CIA-B hingegen steuert die Peripheriegeräte, sowie den Userport. Zusätzlich hierzu erzeugt sie die Interruptsignale, die einen NMI auslösen. Je nach dem ob wir nun IRQs oder NMIs erzeugen möchten, müssen wir also entweder auf CIA-A, oder CIA-B zurückgreifen. Hierbei sei angemerkt, daß wir das natürlich nur dann tun müssen, wenn wir einen timergesteuerten Interrupt programmieren möchten. Innerhalb der CIAs gibt es zwar noch eine ganze Reihe weiterer Möglichkeiten einen Interrupt zu erzeugen, jedoch wollen wir diese hier nicht ansprechen. Hier muß ich Sie auf einen schon vor längerer Zeit in der MD erschienenen CIA-Kurs, in dem alle CIA-Interrupt- quellen ausführlich behandelt wurden, verweisen. Wir wollen uns hier ausschließlich auf die timergesteuerten CIA-Interrupts konzentrieren.
Beide CIAs haben nun jeweils 16 Register, die aufgrund der Gleichheit, bei beiden Bausteinen dieselbe Funktion haben. Einziger Unterschied ist, daß die Register von CIA-A bei $DC00, und die von CIA-B bei $DD00 angesiedelt sind.
Diese Basisadressen müssen Sie also zu dem entsprechenden, hier genannten, Registeroffset hinzuaddieren, je nach dem welche CIA Sie ansprechen möchten. Von den 16 Registern einer CIA sind insgesamt 7 für die Timerprogrammierung zuständig. Die anderen werden zur Datenein- und -ausgabe, sowie eine Echtzeituhr verwandt und sollen uns hier nicht interessieren.
In jeder der beiden CIAs befinden sich nun zwei 16- Bit-Timer, die man mit Timer A und B bezeichnet. Beide können getrennt voneinander laufen, und getrennte Interrupts erzeugen, oder aber zu einem einzigen 32-Bit-Timer kombiniert werden.
Was tut nun so ein Timer? Nun, prinzipiell kann man mit ihm bestimmte Ereignisse zählen, und ab einer bestimmten Anzahl dieser Ereignisse von der dazugehörigen CIA einen Interrupt auslösen lassen. Hierzu hat jeder der beiden Timer zwei Register, in denen die Anzahl der zu zählenden Ereignisse in Low/ High-Byte- Darstellung geschrieben wird. Von diesem Wert aus zählt der Timer bis zu einem Unterlauf (Zählerwert=0), und löst anschließend einen Interrupt aus.

Hier eine Liste mit den 4 Zählerregistern:

Register	Name	Funktion
4	TALO	Low-Byte Timerwert A
5	TAHI	High-Byte Timerwert A
6	TBLO	Low-Byte Timerwert B
7	TBHI	High-Byte Timerwert B

Schreibt man nun einen Wert in diese Timerregister, so wird selbiger in ein internes "Latch"-Register übertragen und bleibt dort bis zu nächsten Schreibzugriff auf das Register erhalten. Auf diese Weise kann der Timer nach einmaligem Herunterzählen, den Zähler wieder mit dem Anfangswert initialisieren.
Liest man ein solches Register aus, so erhält man immer den aktuellen Zählerstand. Ist der Timer dabei nicht gestoppt, so bekommt man jedesmal verschiedene Werte.
Zusätzlich gibt es zu jedem Timer auch noch ein Kontrollregister, in dem festgelegt wird, welche Ereignisse gezählt werden sollen. Weiterhin sind hier Kontrollfunktionen untergebracht, mit denen man den Timer z.B. starten und stoppen kann. Auch hier gibt es einige Bits, die für uns irrelevant sind, weswegen ich sie hier nicht nenne. Das Kontrollregister für Timer A heißt "CRA" und liegt an Registeroffset 14, das für Timer B heißt "CRB" und ist im CIA-Register 15 untergebracht. Hier nun die Bit-Belegung von CRA:

Bit 0 (START/ STOP) Mit diesem Bit schalten Sie den Timer an (=1) oder aus (=0).
Bit 3 (ONE-SHOT/ CONTINOUS) Hiermit wird bestimmt, ob der Timer nur ein einziges Mal zählen, und dann anhalten soll (=1), oder aber nach jedem Unterlauf wieder mit dem Zählen vom Anfangswert aus beginnen soll.
Bit 4 (FORCE LOAD) Ist dieses Bit bei einem Schreibvorgang auf das Register gesetzt, so wird das Zählregister, unabhängig, ob es gerade läuft oder nicht, mit dem Startwert aus dem Latch-Register initialisiert.
Bit 5 (IN MODE) Dieses Bit bestimmt, welche Ereignisse Timer A zählen soll. Bei gesetztem Bit werden positive Signale am CNT-Eingang der CIA gezählt. Da das jedoch nur im Zusammenhang mit einer Hardwareerweiterung einen Sinn hat, lassen wir das Bit gelöscht. In dem Fall zählt der Timer nämlich die Taktzyklen des Rechners.

Dies ist generell auch unsere Arbeitsgrundlage, wie Sie weiter unten sehen werden.
Kommen wir nun zur Beschreibung von CRB (Reg.15) . Dieses Register ist weitgehend identisch mit CRA, jedoch unterscheiden sich Bit 5 und 6 voneinander.
Diese beiden Bits bestimmen nämlich ZUSAMMEN, die Zählerquelle für Timer B (IN MODE). Aus den vier möglichen Kombinationen sind jedoch nur zwei für uns interessant. Setzt man beide Bits auf 0, so zählt Timer B wieder Systemtaktimpulse. Setzt man Bit 6 auf 1 und Bit 5 auf 0, so werden Unterläufe von Timer A gezählt. Auf diese Art und Weise kann man beide Timer miteinander koppeln, und somit Zählerwerte verwenden, die größer als $FFFF sind (was der Maximalwert für ein 16-Bit-Wert ist).
Nun wissen wir also, wie man die beiden Timer initialisieren kann, und zum Laufen bringt. Es fehlt nun nur noch ein Register, um die volle Kontrolle über die CIA-Timer zu haben. Es heißt "Interrupt-Control-Register" ("ICR") und ist in Register 13 einer CIA untergebracht. Mit ihm wird angegeben, welche CIA-Ereignisse einen Interrupt erzeugen sollen. Auch hier sind eigentlich nur drei Bits für uns von Bedeutung. Die Restlichen steuern andere Interruptquellen der CIA, die uns im Rahmen dieses Kurses nicht interessieren sollen.
Es sei angemerkt, daß der Schreibzugriff auf dieses Register etwas anders funk- tioniert als sonst. Will man nämlich bestimmte Bits setzen, so muß Bit 7 des Wertes, den wir schreiben möchten, ebenfalls gesetzt sein. Alle anderen Bits werden dann auch im ICR gesetzt. Die Bits, die im Schreibwert auf 0 sind, beeinflussen den Registerinhalt nicht.
So kann z. B. Bit 0 im ICR schon gesetzt sein. Schreibt man nun den Binärwert 10000010 (=$81) in das Register, so wird zusätzlich noch Bit 1 gesetzt. Bit 0 bleibt davon unberührt, und ebenfalls gesetzt (obwohl es im Schreibwert gelöscht ist!). Umgekehrt, werden bei gelöschtem 7 . Bit alle gesetzten Bits des Schreibwertes im ICR gelöscht. Um also Bit 0 und 1 zu löschen müsste der Binärwert 00000011 geschrieben werden.
Näheres dazu finden Sie in einem Beispiel weiter unten.
Die nun für uns relevanten Bits sind die schon angesprochenen Bits 0,1 und 7.
Die Funktion des 7 . Bits sollte Ihnen jetzt ja klar sein. Bit 0 und 1 geben an, ob Timer A

oder Timer B (oder beide) einen Interrupt auslösen sollen. Sie müssen das entsprechende Bit lediglich auf die oben beschriebene Art setzen, um einen entsprechenden Interrupt zu erlauben. Um z.B. einen Timer-A-Unterlauf als Interruptquelle zu definieren, müssen Sie den Wert $81 in das ICR schreiben.
Für einen Timer-B-Unterlauf $82. Für beide Timer als Interruptquelle $83.
Das ICR hat nun noch eine weitere Funktion. Tritt nämlich ein Interrupt auf, so wissen wir als Programmierer ja noch nicht, ob es tatsächlich ein CIA-Interrupt war, da es auch moch andere Interruptquellen als nur die CIA gibt.
Um nun zu überprüfen, ob der Interrupt von einer CIA stammt, kann das ICR ausgelesen werden. Ist in diesem Wert nun das 7. Bit gesetzt, so heißt das, das eines der erlaubten Interruptereignisse eingetreten ist. Wenn wir wissen möchten, um welches Ereignis es sich dabei genau handelt, brauchen wir nur die Bits zu überprüfen, die die Interruptquelle angeben. Ist Bit 0 gesetzt, so war es Timer A, der den Interrupt auslöste, ist Bit 1 gesetzt, so kam die Unterbrechung von Timer B. Das Auslesen des ICR hat übrigens noch eine weitere Funktion:
solange in diesem Register ein Interrupt gemeldet ist, werden weitere Interruptereignisse ignoriert. Erst wenn das Register ausgelesen wird, wird der CIA signalisiert, daß der Interrupt verarbeitet wurde und neue Unterbrechungen erlaubt sind. Auf diese Weise kann verhindert werden, daß während der Abarbeitung eines Interrupts noch ein zweiter ausgelöst wird, was womöglich das gesamte Interruptsystem durcheinander bringen könnte. Sie müssen also, egal ob Sie sicher sind, daß der Interrupt von der CIA kam, oder nicht - das ICR immer einmal pro Interrupt auslesen, damit der Nächste ausgelöst werden kann. Beachten Sie dabei auch, daß Sie das Register mit dem Auslesen gleichzeitig auch löschen!
Sie können den gelesenen Wert also nicht zweimal über das Register abfragen!
Nach all der trockenen Theorie, wollen wir einmal in die Praxis übergehen und uns einem Programmbeispiel widmen. Wir wollen einmal ein Sprite mittels Joystick in Port 2 über den Bildschirm bewegen. Die Abfrage desselben soll im NMI geschehen, wobei wir CIA-B 30 Mal einen Timerinterrupt pro Sekunde auslösen lassen. Timer A soll für diese Aufgabe herhalten. Hier das Programmlisting des Beispiels, das Sie auf dieser MD auch unter dem Namen "NMI-SPR- DEMO" finden:

```
;Initialisierung
                1000:   LDA #$01        ;Sprite 0
                1002:   STA $D015       ;einschalten und
                1005:   STA $D027       ;Farbe auf Weiß
                1008:   LDA #$3F        ;Sprite-Pointer auf
                100A:   STA $07E8       ;$3F*$40=$0FC0
                100D:   LDA #$64        ;X-/Y-Position des
                100F:   STA $D000       ;Sprites auf
                1012:   STA $D001       ;100/100 setzen
                1015:   LDX #$0E        ;Rahmen- und Hintergrundfarbe
                1017:   LDY #$06        ;auf
                1019:   STX $D020       ;Hellblau und
                101C:   STY $D021       ;Dunkelblau
                101F:   LDA #$7F        ;CIA-B-Interrupt-Quellen
                1021:   STA $DD0D       ;sperren
                1024:   LDA #$00        ;Timer A von CIA-B
                1026:   STA $DD0E       ;anhalten.
                1029:   LDX #$48        ;Startadresse für neuen
                102B:   LDY #$10        ;NMI (=$1048) im
                102D:   STX $0318       ;Pointer bei $0318/0319
                1030:   STY $0319       ;ablegen.
                1033:   LDX #$49        ;Timerzählwert=$8049
                1035:   LDY #$80        ;(30 Mal pro Sekunde)
```

```
                1037:   STX $DD04       ;in Timerzählerregister
                103A:   STY $DD05       ;(Timer A) schreiben
                103D:   LDA #$81        ;Timer A als Interrupt-
                103F:   STA $DD0D       ;quelle erlauben.
                1042:   LDA #$11        ;Timer starten (mit
                1044:   STA $DD0E       ;FORCE-LOAD-Bit)
                1047:   RTS             ;ENDE
;Interruptroutine
                1048:   PHA             ;Akku retten
                1049:   TXA             ;X in Akku
                104A:   PHA             ;und retten
                104B:   TYA             ;Y in Akku
                104C:   PHA             ;und retten
                104D:   LDA $DD0D       ;ICR auslesen
                1050:   BNE $1058       ;Wenn<>0, dann CIA-NMI
                1052:   INC $D020       ;Sonst NMI von RESTORE-Taste / Rahmenf. erh.
                1055:   JMP $1073       ;und auf Ende NMI springen
                1058:   LDA $DC01       ;CIA-NMI: Joyport2 auslesen
                105B:   LSR             ;Bit in Carry-Flag rot.
                105C:   BCS $1061       ;Wenn gelöscht Joy hoch
                105E:   DEC $D001       ;SpriteX-1
                1061:   LSR             ;Bit in Carry-Flag rot.
                1062:   BCS $1067       ;Wenn gel., Joy runter
                1064:   INC $D001       ;SpriteX+1
                1067:   LSR             ;Bit in Carry-Flag rot.
                1068:   BCS $106D       ;Wenn gel. Joy rechts
                106A:   INC $D000       ;SpriteY+1
                106D:   LSR             ;Bit in Carry-Flag rot.
                106E:   BCS $1073       ;Wenn gel. Joy links
                1070:   DEC $D000       :SpriteY-1
                1073:   PLA             ;Y-Reg vom Stapel
                1074:   TAY             ;holen
                1075:   PLA             ;X-Reg vom Stapel
                1076:   TAX             ; holen
                1077:   PLA             ;Akku vom Stapel holen
                1078:   CLI             ;IRQs wieder erlauben
                1079:   RTI             ;RETURN FROM INTERRUPT
;Eigenen NMI entfernen
                107A:   LDA #$7F        ;Alle NMI-Quellen von
                107C:   STA $DD0D       ;CIA-B sperren
                107F:   LDA #$00        ;Timer A von CIA-B
                1081:   STA $DD0E       ;stoppen
                1084:   LDX #$47        ;Alten NMI-Vektor
                1086:   LDY #$FE        ;(bei $FE47) wieder in
                1088:   STX $0318       ;$0318/$0319
                108B:   STY $0319       ;eintragen
                108E:   LDA #00         ;Sprite 0
                1090:   STA $D015       ;ausschalten
                1093:   RTS             ;ENDE
```

Das Programm starten Sie mit SYS4096.
Hieraufhin sehen Sie einen Sprite-Pfeil auf dem Bildschirm, der mit einem Joystick in Port 2 über den Bildschirm bewegt werden kann. Gleichzeitig ist jedoch weiterhin die normale Tastaturabfrage des C64 aktiv. Die beiden Prozesse laufen scheinbar gleichzeitig ab, weil wir einen zweiten Interrupt generiert haben, der den Joystick abfragt und entsprechend das Sprite bewegt. Kommen wir nun zur Programmbeschreibung:
In den ersten Zeilen, von $1000 bis $101E wird zunächst einmal das Sprite initialisiert und die Bildschirmfarben auf die gängige Kombination hellblau/ dunkelblau gesetzt. Hiernach

folgt die NMI und Timerinitialisierung. Da wir auch hier verhindern müssen, daß ein NMI auftritt, während wir den NMI-Vektor verändern, wird der Wert $7F in das ICR geschrieben. Dieser Wert löscht alle Interruptquellen-Bits, womit kein NMI mehr von CIA-B ausgelöst werden kann. Da sie der einzige Chip ist, der NMIs auslöst, können wir sicher sein, daß tatsächlich keiner dieser Interrupts aufgerufen wird. Beachten Sie bitte, daß der SEI-Befehl hier wirkungslos wäre, denn er sperrt lediglich die IRQs. NMIs sind nicht abschaltbar, weswegen wir sie auf diese umständliche Art und Weise unterbinden müssen.
Hiernach halten wir Timer A an, indem wir einen Wert in CRA schreiben, der das 0. Bit gelöscht hat, woraufhin die CIA den Timer stoppt. Dies müssen wir tun, damit der Timer korrekt mit seinem Anfangszählwert gefüttert werden kann. Würden wir nämlich das Low-Byte geschrieben haben, das High-Byte jedoch noch nicht, während ein Unterlauf des Timers stattfindet, so würde er mit sich mit einem falschen Anfangswert initialisieren.
In den folgenden Zeilen wird nun der NMI-Vektor bei $0318/$0319 auf unsere eigene NMI-Routine bei $1048 verbogen.
Hiernach wird der Timer mit seinem Zählwert initialisiert. Er beträgt $8049.
Wie wir auf diesen Wert kommen, möchte ich Ihnen nun erläutern: Wir hatten ja schon festgestellt, daß wir den Timer vorwiegend die Taktzyklen des 64ers zählen lassen möchten. Selbige sind der Taktgeber für den Prozessor. Der Quarz der sie erzeugt generiert zusammen mit einigen Bausteinen um ihn herum einen Takt von genau 985248,4 Impulsen pro Sekunde, die auch in Herz (Hz) gemessen werden. Dieser Wert entspricht also annähernd einem MHz. Um nun zu ermitteln, wie viele Taktzyklen gezählt werden müssen, damit der Timer genau 30 Mal pro Sekunde einen Interrupt erzeugt, müssen wir diesen Wert lediglich durch 30 dividieren. Das Ergebnis hiervon ist genau 32841,6, was $8049 entspricht! Um nun z.B.50 Interrupts pro Sekunde zu erzeugen dividieren Sie einfach durch 50, usw. Auf diese Weise können Sie die Joystickabfrage auch beschleunigen oder verlangsamen, denn das Sprite wird bei betätigtem Joystick immer um genau soviele Pixel pro Sekunde verschoben, wie Interrupts auftreten. Versuchen Sie die Timerwerte doch einmal auf 100 oder nur 20 Interrupts pro Sekunde verändern!
Nachdem der Timer nun mit seinem Startwert gefüttert wurde, erlauben wir den Timer-A-Interrupt durch Setzen des 0. Bits im ICR. Hiernach muß der Timer nur noch gestartet werden. Damit er gleich beim richtigen Wert beginnt zu zählen, ist hier außer dem START/STOP-Bit auch noch das FORCE-LOAD-Bit gesetzt. Das 3. Bit ist auf 0, damit der Timer im CONTINOUS-Modus läuft. Ebenso wie das 5. Bit, das ihm sagt, daß er Taktzyklen zählen soll. Alles in allem wird also der Wert $11 in CRA geschrieben. Von nun an läuft der Timer und wird bei einem Unterlauf einen Interrupt erzeugen, der unsere NMI-Routine bei $1048 anspringen wird.
Kommen wir nun zu dieser Routine selbst.
Zu Anfang müssen wir erst einmal die drei Prozessorregister retten, da wir ja einen NMI programmiert haben, bei dem uns die Betriebsystemsvorbeitungsroutine diese Arbeit noch nicht abgenommen hat (im Gegensatz zu der IRQ-Routine). Hiernach wird das ICR von CIA-B ausgelesen.
Dadurch ermöglichen wir es diesem Chip beim nächsten Timerunterlauf erneut einen Interrupt auslösen zu können.
Gleichzeitig können wir dadurch abfragen, ob der NMI tatsächlich von der CIA kam. Dies ist zwar sehr wahrscheinlich, da die CIA der einzige Chip ist, der diese Art von Interrupt erzeugen kann, jedoch ist die RESTORE-Taste auf der Tastatur des C64 ebenso mit dem NMI-Eingang des Prozessors verbunden. Das heißt also, daß durch Drücken dieser Taste ebenfalls ein NMI ausgelöst wird.
In dem Fall erhalten wir beim Auslesen des ICR den Wert 0, da ja kein CIA-Interrupt auftrat. Dadurch können wir abfragen, ob es sich um unseren Timer-NMI, oder einen RESTORE-Tastendruck handelt. Ist letzterer der Auslöser, so wird einfach die Rahmenfarbe um eins erhöht, und zum Ende des NMIs weitergeprungen. Handelt es sich

um einen CIA-NMI, so wird direkt zur Joystickabfrage weiter verzweigt, in der wir Bitweise die Portbits von CIA-A auswerten.
Übrigens: da in dem gelesenen Register auch Datenbits der Tastatur erscheinen, kann es passieren, daß Sie bei bestimmten Tastendrücken ebenfalls das Sprite über den Bildschirm zucken sehen. Dies ist normal und handelt sich nicht um einen Fehler!
Zum Abschluß der Routine holen wir die drei Prozessorregister wieder vom Stapel und erlauben mittels CLI die IRQs wieder. Selbige wurden nämlich in der Betriebsystemsvorbeitung zum NMI abgeschaltet (sh. Teil1 dieses Kurses). Das Programm wird diesmal nicht mit einem Sprung auf die NMI-Behandlungsroutine des Betriebssystems beendet, da diese ausschließlich für die Behandlung der RESTORE-Taste benutzt wird und demnach bei einfachem Drücken der RUN-STOP- Taste sofort einen BASIC-Warmstart auslösen würde (so wird nämlich die Tastenkombination RUN-STOP/ RESTORE abgefragt).
Am Ende des Programmbeispiels sehen Sie noch eine Routine, die unsere NMI-Routine wieder aus dem System entfernt.
Sie sperrt wieder alle NMI-Interrupts von CIA-B und stoppt ihren Timer A. Anschließend wird der NMI-Vektor bei $0318/$0319 wieder auf die ursprüngliche Adresse, nämlich $FE47, zurückgebogen.
Das war es nun für diesen Monat. Probieren Sie ein wenig mit den Timerinterrupts herum, und versuchen Sie z.B. einen solchen Interrupt von Timer B auslösen zu lassen. Prüfen Sie auch einmal, was passiert, wenn Sie andere Timerwerte benutzen. Ich empfehle Ihnen, eine Routine zu schreiben, die immer nach genau $4CC7 Taktzyklen ausgelöst werden soll und dann für eine kurze Zeit die Rahmenfarbe verändert. Sie werden hierbei schon einmal einen Vorgeschmack auf den nächsten Kursteil bekommen, in dem wir uns endlich dann den Rasterinterrupts nähern werden.

(ub)

Teil 3 – Magic Disk 01/94

Herzlich willkommen zum dritten Teil unseres Interruptkurses. In diesem Monat möchten wir endlich in "medias res" gehen und in den interessantesten Teilbereich der Interruptprogrammierung einsteigen: den Raster-Interrupt.

1. Was ist ein Raster-Interrupt

Raster-Interrupts sind Interrupts die vom Videochip des C64 ausgelöst werden.
Er ist die dritte hardwaremäßige Interruptquelle und bietet uns vielfältige Möglichkeiten um besonders interessante und schöne Interrupt-Effekte erzeugen zu können. Zunächst jedoch wollen wir klären, was so ein Raster-Interrupt nun eigentlich ist. Hierzu wollen wir zuerst einmal die Funktionsweise eines Computermonitors oder Fernsehers in groben Zügen besprechen:
Der Bildschirm, so wie Sie ihn jetzt vor sich haben, ist an seiner Innenseite mit einem Stoff beschichtet, der, wenn er von Elektronen getroffen wird, zum Leuchten angeregt wird. Innerhalb des Monitors befindet sich nun eine Elektronenquelle, die, durch Magnetfelder gebündelt, einen hauchdünnen Strahl von Elektronen erzeugt. Zusätzlich gibt es noch zwei Ablenkmagnete, die den Strahl in der Horizontalen und Vertikalen ablenken können, so daß jede Position auf dem Bildschirm erreicht werden kann.
Der Elektronenstrahl wird nun durch ein bestimmtes Bewegungsschema zeilenweise über den Bildschirm bewegt und bringt selbigen zum Leuchten. Hierbei ist der gesamte Bildschirm in 313 sogenannte Rasterzeilen aufgeteilt. Jedes mal, wenn eine Zeile von links nach rechts aufgebaut wurde, wird der Stahl kurzfristig abgeschaltet und am linken Bildschirmrand wieder angesetzt, um die nächste Zeile zu zeichnen. Dies geschieht mit

einer unglaublichen Geschwindigkeit, so daß für das sehr träge Auge ein Gesamtbild aus unzähligen Leuchtpunkten auf dem Bildschirm entsteht. Insgesamt 25 Mal pro Sekunde 'huscht' der Elektronenstahl auf diese Weise über den Bildschirm, wodurch 25 Bilder pro Sekunde aufgebaut werden. Damit nun ein ganz spezielles Videobild auf dem Bildschirm zu sehen ist, z. B. der Text, den Sie gerade lesen, benötigt man eine Schaltlogik, die die Text oder Grafikzeichen aus dem Speicher des Computers in sichtbare Bildpunkte umwandelt. Diese Umwandlung geschieht durch den Video-Chip des C64, dem sogenannten VIC. Er erzeugt Steuersignale für den Rasterstrahl, die angeben in welcher Rasterzeile er gerade arbeiten soll, und mit welcher Intensität die einzelnen Bildpunkte auf dem Bildschirm leuchten sollen. Helle Bildpunkte werden dabei mit mehr, dunklere mit weniger Elektronen 'beschossen'. Da der VIC nun auch die Steuersignale für den Rasterstrahl erzeugt, weiß er auch immer, wo sich selbiger gerade befindet.
Und das ist der Punkt an dem wir einsetzen können, denn der VIC bietet uns nun die Möglichkeit, diese Strahlposition in einem seiner Register abzufragen. Damit wir nun aber nicht ständig prüfen müssen, in welcher der 313 Rasterzeilen sich der Strahl nun befindet, können wir ihm auch gleich mitteilen, daß er einen Interrupt bei Erreichen einer bestimmten Rasterzeile auslösen soll. Dies geschieht über spezielle Interrupt-Register, deren Belegung ich Ihnen nun aufführen möchte.

2. Die Raster-IRQ-Programmierung

Zunächst einmal brauchen wir die Möglichkeit, die gewünschte Rasterzeile festlegen zu können. Dies wird in den Registern 17 und 18 (Adressen $D011 und $D012) des VICs angegeben. Da es ja insgesamt 313 Rasterzeilen gibt reicht nämlich ein 8-Bit-Wert nicht aus, um alle Zeilen angeben zu können. Aus diesem Grund ist ein zusätzliches neuntes Bit in VIC-Register 17 untergebracht. Bit 7 dieses Registers gibt zusammen mit den acht Bits aus VIC-Register 18 nun die gewünschte Rasterzeile an. Da die anderen Bits dieses Registers ebenfalls einige Aufgaben des VICs ausfüllen, dürfen wir nur durch logische Verknüpfungen darauf zugreifen. Möchten wir also eine Rasterzeile von 0 bis 255 als Interruptauslöser definieren, so muß das 7. Bit mit den folgenden drei Befehlen gelöscht werden. Die Rasterzeilen-Nummer wird dann ganz normal in Register 18 ($D012) geschrieben:

```
LDA $D011          ;Reg.17 lesen
AND #$7F           ;7. Bit löschen
STA $ D011         ;Reg.17 zurückschreiben
```

Wenn nun eine Rasterzeile größer als 255 in die beiden VIC-Register geschrieben werden soll, so müssen wir mit den folgenden drei Befehlen das 7. Bit in Register $D011 setzen, und die Nummer der Rasterzeile minus dem Wert 256 in Register $D012 schreiben:

```
LDA $D011          ;Reg.17 lesen
ORA $80            ;7. Bit setzen
STA $D011          ;Reg.17 zurückschreiben.
```

Um z.B. Rasterzeile 300 als Interruptauslöser einzustellen, müssen wir wie eben gezeigt das 7. Bit in $D011 setzen und anschließend den Wert 300-256=44 in $D012 ablegen.
Das alleinige Festlegen einer Rasterzeile als Interruptauslöser genügt jedoch noch nicht, den VIC dazu zu bringen den Prozessor beim Erreichen dieser Zeile zu unterbrechen. Wir müssen ihm zusätzlich noch mitteilen, daß er überhaupt einen Interrupt auslösen soll. Dies wird in Register 26 (Adresse $D01A) festgelegt.
Es ist das Interrupt-Control-Register (ICR) des VIC und hat eine ähnliche Funktion wie das ICR der CIA, das wir im letzten Kursteil schon kennengelernt hatten.
Das VIC-ICR kennt vier verschiedene Ereignisse, die den VIC zum Auslösen eines Interrupts bewegen. Jedes der Ereignisse wird durch ein zugehöriges Bit im ICR repräsentiert, das lediglich gesetzt werden muß, um das entsprechende Ereignis als

Interruptquelle zu definieren. Hier die Belegung:

Bit	Interruptquelle
0	Rasterstrahl
1	Kollision Sprites und Hintergrund
2	Kollision zwischen Sprites
3	Lightpen sendet Signal

Wie Sie sehen ist für uns hauptsächlich Bit 0 von Bedeutung. Setzen wir es, so löst der VIC bei Erreichen der zuvor festgelegten Rasterzeile einen Interrupt aus. Sie sehen sicherlich auch, daß Sprite-Kollisionen mit Hintergrundgrafik und/ oder mit anderen Sprites ebenso einen Interrupt auslösen können. Sie können auf diese Weise also auch sehr komfortabel eine Kollision über den Interrupt abfragen. Bit 3 wird nur sehr selten benutzt und bringt eigentlich nur etwas in Zusammenhang mit einem Lightpen. Selbiger sendet nämlich immer dann ein Signal, wenn der Rasterstrahl gerade an ihm vorübergezogen ist. Wird so nun ein Interrupt ausgelöst, muß das entsprechende Lightpen-Programm nur noch auswerten, an welcher Rasterposition der Strahl sich gerade befindet, um herauszufinden, an welcher Stelle des Bildschirms sich der Lightpen befindet. Dies soll uns hier jedoch nicht interessieren.
Wir müssen also lediglich Bit 0 von $D01A durch Schreiben des Wertes 1 setzen, um Raster-Interrupts auslösen zu können.

3. Das Handling der Raster-Interrupts

Hardwaremäßig haben wir nun alles festgelegt und den VIC auf die Interrupt-Erzeugung vorbereitet. Was nun noch fehlt ist die Software die den Interrupt bedient. Auch dies ist für uns im Prinzip ein alter Hut, denn der VIC ist mit der IRQ-Leitung des Prozessors verbunden, weswegen wir einen Raster-Interrupt im Prinzip wie den CIA1- Interrupt des Betriebssystems abfragen können. Hierzu brauchen wir natürlich erst einmal eine Initialisierungsroutine, die den IRQ-Vektor bei $0314/$0315 auf unsere eigene Interrupt-Routine verbiegt. Hierbei müssen wir jedoch beachten, daß der Betriebssystem- Interrupt, der von Timer A der CIA1 ausgelöst wird, ebenfalls über diesen Vektor springt. Das bedeutet, daß unsere Routine zwischen diesen beiden Interrupts unterscheiden muß, da ein Raster-IRQ anders bedient werden muß, als ein CIA-IRQ. Würden wir keinen Unterschied machen, so würde uns die CIA ungehemmt 'dazwischenfunken' und der Rasterinterrupt würde außer Kontrolle geraten. Bevor ich mich nun aber in theoretischen Fällen verliere, sehen Sie sich einmal folgendes Beispielprogramm an, anhand dessen wir die Lösung dieses Problems besprechen möchten:

```
;*** Initialisierung
        Init:       SEI             ;IRQs sperren
                    LDA #$00        ;Rasterzeile 0 als Inter-
                    STA $d012       ;ruptauslöser festlegen
                    LDA $d011       ;(höchstwertiges Bit
                    AND #$7F        ;der Rasterzeile
                    STA $d011       ;löschen)
                    LDA #$01        ;Rasterstrahl als Inter-
                    STA $d01A       ;ruptquelle festlegen)
                    LDA #3          ;Zähler in $02 mit dem
                    STA $02         ;Wert 3 initialisieren)
                    LDX #<(irq)     ;IRQ-Vektor bei $0314/
                    LDY #>(irq)     ;$0315 auf eigene
                    STX $0314       ;Routine mit dem Namen
                    STY $0315       ;"IRQ" verbiegen.
```

```
                    CLI             ;IRQs wieder erlauben
                    RTS             ;ENDE!
;*** Interrupt-Routine
        irq:        LDA $D019       ;VIC-IRR lesen
                    STA $D019       ;und zurückschreiben
                    BMI raster      ;Wenn 7.Bit gesetzt, war es ein Raster-IRQ
                    JMP $EA31       ;Sonst auf Betriebssystem
; Routine springen
        raster:     DEC $02         ;Zähler runterzählen
                    LDA $02         ;Zähler lesen
                    CMP #2          ;Zähler=2 ?
                    BNE rast70      ;Nein, also weiter
                    LDA #$70        ;Sonst Zeile $70 als neu-
                    STA $D012       ;n Auslöser festlegen
                    LDA #$00        ;Rahmen- und Hintergrund-
                    STA $D020       ;farbe auf 'schwarz'
                    STA $D021       ;setzen
                    JMP $FEBC       ;IRQ beenden
        rast70:     CMP #1          ;Zähler=1 ?
                    BNE rastc0      ;Nein, also weiter
                    LDA #$c0        ;Sonst Zeile $C0 als neu-
                    STA $d012       ;en Auslöser festlegen
                    LDA #$02        ;Rahmen- und Hintergrund-
                    STA $d020       ;farbe auf 'rot'
                    STA $d021       ;setzen
                    JMP $FEBC       ;IRQ beenden
        rastc0:     LDA #$00        ;Wieder Zeile 0 als Aus
                    STA $D012       ;löser festlegen
                    LDA #$07        ;Rahmenund Hintergrund
                    STA $D020       ;farbe auf ' gelb'
                    STA $D021       ;setzen
                    LDA #3          ;Zähler wieder auf 3
                    STA $02         ;zurücksetzen
                    JMP $FEBC       ;IRQ beenden
```

Sie finden das Programm auch als ausführbaren Code unter dem Namen "RASTERDEMO1" auf dieser MD. Es muß absolut (mit ",8,1") geladen und mit "SYS4096" gestartet werden. Möchten Sie es sich ansehen, so disassemblieren Sie mit einem Monitor ab Adresse $1000.
Unser erster Raster-Interrupt soll nun auf dem Bildschirm eine Schwarz-Rot-Gold-Flagge darstellen. Zu diesem Zweck lassen wir den VIC in den drei Rasterzeilen $00, $70 und $C0 einen Interrupt auslösen, woraufhin wir die Rahmen und Hintergrundfarbe in eine der drei Farben ändern. Gehen wir jedoch Schritt für Schritt vor und schauen wir uns zunächst einmal die Initialisierungsroutine an:
Hier schalten wir zunächst einmal die IRQs mittels des SEI-Befehls ab, und setzen anschließend auf die schon beschriebene Art und Weise die Rasterzeile 0 als Interruptauslösende Zeile fest. Hieraufhin wird der Wert $01 in Register $D01A geschrieben, womit wir die Rasterinterrupts erlauben. Danach wird noch ein Zähler in Adresse $02 initialisiert, den wir zum Abzählen der Raster-Interrupts benötigen, damit wir auch immer die richtige Farbe an der richtigen Rasterposition setzen. Zuletzt wird der IRQ-Vektor bei $0314/$0315 auf unsere Routine verbogen und nach einem abschließenden Freigeben der IRQs wird die Initialisierung beendet. Von nun an wird bei jedem IRQ, stammt er nun vom VIC oder der CIA, die den Betriebssystem IRQ erzeugt, auf unsere Routine "IRQ" gesprungen.
Damit der Raster-Interrupt nun jedoch nicht, wie oben schon angesprochen, außer Kontrolle gerät, müssen wir hier nun als erstes herausfinden, von welchem der beiden

Chips der IRQ ausgelöst wurde. Dies geschieht durch Auslesen des Registers 25 des VICs (Adresse $D019).
Es das Interrupt-Request-Register dieses Chips, in dem das 7. Bit, sowie das entsprechende Bit des ICR, immer dann gesetzt werden, wenn eines der vier möglichen Interrupt-Ereignisse des VICs eingetreten ist. Durch das Lesen dieses Registers holen wir uns also seinen Wert in den Akku. Das Anschließende zurückschreiben löscht das Register wieder.
Dies ist notwendig, da der VIC den soeben in diesem Register erzeugten Wert solange hält, bis ein Schreibzugriff darauf durchgeführt wird. Dadurch wird sichergestellt, daß die Interruptanforderung auch tatsächlich bei einem behandelnden Programm angelangt ist. Würden wir nicht schreiben, so würde der VIC den nächsten Interrupt schlichtweg ignorieren, da er den alten ja noch für nicht abgearbeitet interpretiert. Der Schreibzugriff auf das IRR des VICs entspricht im Prinzip also derselben Funktion, wie der Lesezugriff des ICR einer CIA, wenn von dort ein Interrupt ausgelöst wurde.
Im Akku befindet sich nun also der Wert aus dem IRR. Gleichzeitig mit dem Lesevorgang wurden die dem Wert entsprechenden Flags im Statusregister des Prozessors gesetzt. Wir müssen nun also lediglich prüfen, ob das 7. Bit dieses Wertes gesetzt ist, um herauszufinden, ob es sich um einen vom VIC stammenden Raster-IRQ handelt, oder nicht. Dies geschieht durch den folgenden BMI-Befehl, der nur dann verzweigt, wenn ein negativer Wert das Ergebnis der letzten Operation war. Da ein negativer Wert aber immer das 7. Bit gesetzt hat, wird also nur in diesem Fall auf die Unterroutine "RASTER" verzweigt. Im anderen Fall springen wir mit einem " JMP $EA31", die Betriebssystem-IRQ-Routine an.
War das 7. Bit nun also gesetzt, so wird ab dem Label "RASTER" fortgefahren, wo wir zunächst einmal unseren Zähler in $02 um 1 erniedrigen, und ihn dann in den Akku holen. Steht dort nun der Wert 2, so wurde der Interrupt von Rasterzeile 0 ausgelöst, und wir setzen Zeile $70 als nächsten Auslöser fest. Hierbei müssen wir Bit 7 von $D011 nicht extra löschen, da es von der Initialisierungsroutine her ja noch auf 0 ist. Danach werden Rahmen und Hintergrundfarbe auf 'schwarz' gesetzt und der Interrupt durch einen Sprung auf Adresse $FEBC beendet. Hier befindet sich das Ende des Betriebssystems-IRQ-Programms, in dem die Prozessorregister wieder zurückgeholt werden, und mit Hilfe des RTI-Befehls aus dem Interrupt zurückgekehrt wird.
Steht in unserem Zähler bei $02 nun der Wert 1, so wurde der Interrupt von Zeile $70 ausgelöst. Hier setzen wir die Bildschirmfarben nun auf 'rot' und legen Zeile $ C0 als Auslöser des nächsten Raster-Interrupts fest. Tritt selbiger auf, so steht weder 1 noch 2 in unserem Zählregister, weswegen zur Routine "RASTC0" verzweigt wird, wo wir die Rahmenfarbe auf 'gelb' setzen, den Zähler wieder mit seinem Startwert initialisieren und Rasterzeile 0 als nächsten Interruptauslöser festlegen, womit sich der ganze Prozeß wieder von vorne wiederholt.

4. Die Probleme bei Raster-IRQs

Wenn Sie unser eben beschriebenes Beispielprogramm einmal gestartet und angeschaut haben, so wird Ihnen sicher aufgefallen sein, daß die Farbbalken nicht ganz sauber auf dem Bildschirm zu sehen waren. Bei jedem Farbübergang flackerte der Bildschirm am linken Rand.
Dies ist eines der größten Probleme bei der Raster-Interrupt-Programmierung. Dadurch nämlich, daß der Rasterstrahl mit einer Wahnsinnsgeschwindigkeit über den Bildschirm huscht können minimale Verzögerungen im Programm, wie zum Beispiel ein weiterer Befehl, bevor die Farbänderung geschrieben wird, verheerende Folgen haben. In unserem Beispiel werden die beiden Farben nämlich genau dann geändert, wenn sich der Rasterstrahl gerade am Übergang vom Bildschirmrahmen zum Bildschirmhintergrund befindet. Dadurch entstehen die kleinen Farbstreifen am linken Rand, wo die

Rahmenfarbe schon auf den neuen Wert, die Hintergrundfarbe jedoch noch auf dem alten Wert steht. Erst wenn letztere ebenfalls geändert wurde, ist das Bild so wie gewünscht. Zusätzlich benötigt unsere Routine immer Unterschiedlich viel Rechenzeit, da sie einige Branch-Befehle enthält, die je nach Zutreffen der abgefragten Bedingung 2 oder 3 Taktzyklen dauern, weswegen die Farbstreifen unruhig hin und herspringen. Zusätzlich kann es passieren, daß kurzfristig einer der drei Farbbereiche nicht richtig eingeschaltet wird, und dann der ganze Balken flackert. Dieser Fall tritt dann ein, wenn Raster und CIA-Interrupt kurz hintereinander auftreten und ein gerade bearbeiteter Raster-IRQ durch den CIA-IRQ nochmals unterbrochen wird. Die Routine kann also noch erheblich verbessert werden! Stellen Sie doch einmal die Farbänderung jeweils an den Anfang der entsprechenden Unterroutine, und sperren Sie gleichzeitig mittels "SEI" das Auftreten weiterer IRQs. Die Flackereffekte sollten sich dadurch schon um einiges vermindern, aber dennoch werden Sie nicht ganz verschwinden. Das liegt hauptsächlich an unserer Verzweigungsroutine, die zwischen VIC und CIA-IRQ unterscheidet.
Da sie für den Rasterstrahl nicht unerheblich Zeit verbraucht wird immer irgendwo eine Asynchronität festzustellen sein.
Sauberer soll das nun mit unserem nächsten Programmbeispiel gelöst werden.
Hier wollen wir ganz einfach den Betriebssystems-IRQ von CIA1 ganz unterbinden, und ihn quasi über den Raster-IRQ 'simulieren' .Zusätzlich soll die Abfrage nach der Rasterzeile, die den Interrupt auslöste wegfallen, so daß wenn unsere Routine angesprungen wird, immer gleich die gewünschte Farbe eingestellt werden kann. Da wir daraufhin auch keine Abfragen mittels Branch-Befehlen in unserem Interrupt-Programm haben, wird die Routine immer mit der gleichen Laufzeit ablaufen, weswegen unruhiges Zittern wegfallen wird. Hier nun zunächst das Beispielprogramm. Sie finden es auf dieser MD als ausführbaren Code unter dem Namen "RASTERDEMO2" . Es wird ebenso geladen und gestartet wie unser erstes Beispiel:

```
;*** Initialisierung
        Init:       SEI             ;IRQs sperren
                    LDA #$7F        ;Alle Bits im CIA-ICR
                    STA $DC0D       ;löschen (CIA-IRQs)
                    LDA $DC0D       ;CIA-ICR löschen
                    LDA #$00        ;Rasterzeile 0 als Inter-
                    STA $D012       ;ruptauslöser festlegen
                    LDA $D011       ;Bit 7 in $D011
                    AND #$7F        ;löschen.
                    STA $D011
                    LDA #$01        ;Rasterstrahl als Inter-
                    STA $D01A       ;rupt-quelle definieren
                    LDX #<(irq1)    ;IRQ-Vektor bei $0314/
                    LDY #>(irq1)    ;$0315 auf erste
                    STX $0314       ;Interrupt-
                    STY $0315       ;Routine verbiegen
                    CLI             ;IRQs wieder freigeben
                    RTS             ;ENDE!
;*** Erster Interrupt
        irq1:       LDA #$70        ;Zeile $70 als Auslöser
                    STA $D012       ;für nächsten IRQ festlegen
                    DEC $d019       ;VIC-IRR löschen
                    LDX #<(irq2)    ;IRQ-Vektor auf
                    LDY #>(irq2)    ;zweite Interrupt-
                    STX $0314       ;Routine
                    STY $0315       ;verbiegen
                    LDA #$00        ;Rahmen u. Hintergrund
                    STA $d020       ;auf 'schwarz'
                    STA $d021       ;setzen
```

```
                    JMP &FEBC           ;IRQ beenden
;*** Zweiter Interrupt
        irq2:       LDA #$C0            ;Zeile $C0 als Auslöser
                    STA $D012           ;für nächsten IRQ festlegen
                    DEC $D019           ;VIC-IRR löschen
                    LDX #<(irq3)        ;IRQ-Vektor auf
                    LDY #>(irq3)        ;dritte Interrupt-
                    STX $0314           ;Routine
                    STY $0315           ;verbiegen
                    LDA #$00            ;Rahmen und Hintergrund
                    STA $D020           ;auf 'rot'
                    STA $D021           ;setzen
                    JMP &FEBC           ;IRQ beenden
;*** Dritter Interrupt
        irq3:   LDA #$C0            ;Wieder Zeile 0 als Aus-
                    STA $D012           ;löser für IRQ festlegen
                    DEC $D019           ;VIC-IRR löschen
                    LDX #<(irq1)        ;IRQ-Vektor wieder auf
                    LDY #>(irq1)        ;erste Interrupt-
                    STX $0314           ;Routine
                    STY $0315           ;verbiegen
                    LDA #$00            ;Rahmen und Hintergrund
                    STA $D020           ;auf 'gelb'
                    STA $D021           ;setzen
                    JMP $EA31           ;Betr.sys.-IRQ anspringen
```

Die Initialisierungsroutine dieses Beispiels unterscheidet sich kaum von dem des ersten Programms. Wir setzen auch hier Zeile 0 als Auslöser für den ersten IRQ und verbiegen den IRQ-Vektor auf die erste IRQ-Routine mit dem Namen "IRQ1".
Ganz am Anfang jedoch unterbinden wir alle CIA1- IRQs, indem wir durch Schreiben des Wertes $7F alle Bits des CIA-ICRs löschen, womit wir alle von CIA1 auslösbaren IRQs sperren. Das anschließende Auslesen des ICRs dient dem 'freimachen' der CIA, falls dort noch eine Interruptanforderung vorliegen sollte, die nach Freigeben der IRQs ja sofort ausgeführt und so in unsere Raster-IRQ- Routine1 springen würde.
Die Interrupt-Routine selbst wird nun wirklich nur dann aufgerufen, wenn sich der Rasterstrahl in Zeile 0 befindet.
Deshalb können wir hier gleich, ohne große Abfragen durchführen zu müssen, Zeile $70 als nächsten Interruptauslöser festlegen und die Bildschirmfarben ändern. Damit hier nun ebenfalls die richtige Routine angesprungen wird, verbiegen wir den IRQ-Vektor noch auf die Routine "IRQ2", die speziell den Interrupt bei Zeile $70 behandeln soll.
Sie verfährt genauso mit dem dritten Interruptauslöser, Rasterzeile $C0, für die die Routine "IRQ3" zuständig ist.
Sie biegt den IRQ-Vektor nun wieder zurück auf "IRQ1", womit das ganze von vorne beginnt. Eine Neuheit ist hier übrigens das Löschen des VIC-IRRs. Um selbiges zu tun, hatten wir in letztem Beispiel ja einen Schreibzugriff auf das Register durchführen müssen. Dies tun wir hier mit dem Befehl "DEC $D019". Er hat den Vorteil daß er kürzer und schneller ist als einzelnes ein Lesen und Schreiben des Registers und zudem kein Prozessorregister in Anspruch nimmt, in das wir lesen müssen.
Wie Sie nun sehen werden ist unsere Flagge nun nicht mehr von Flackereffekten gebeutelt. Die Farbbalken sind sauber durch horizontale Linien voneinander getrennt. Versuchen Sie jetzt doch einmal ein paar andere Farbkombinationen, oder vielleicht auch mehr Rasterbalken zu erzeugen. Oder kombinieren Sie doch einmal Hiresgrafik mit Text. Die Möglichkeiten solch recht einfacherer Raster-IRQs sind sehr vielseitig, und warten Sie erst einmal ab, wenn wir zu komplizierteren Routinen kommen. Dies wird im nächsten Monat dann schon der Fall sein, wenn wir per Rasterstrahl die Bildschirmränder

abschalten werden. Bis dahin viel Spaß beim Rasterprogrammieren!

(ub)

Teil 4 – Magic Disk 02/94

Herzlich Willkommen zum vierten Teil unseres IRQ-Kurses. In dieser Ausgabe möchten wir uns mit sehr zeitkritischen Rastereffekten beschäftigen und kurz zeigen, wie man den oberen und unteren Bildschirmrand mit Hilfe von Rasterinterrupts verschwinden lassen kann.

1. Die TOP- und BOTTOM-Border-Routine

Wie Sie ja sicherlich wissen, ist der Bildschirm des C64 auf einen sichtbaren Bereich von 320 x 200 Pixeln, oder auch 40 x 25 Textzeichen beschränkt. Der Rest des Bildschirms ist von dem meist hellblauen Bildschirmrahmen verdeckt und kann in der Regel nicht genutzt werden.
Werfen Sie jetzt jedoch einen Blick auf diese Zeilen, so werden Sie feststellen, daß das Magic-Disk-Hauptprogramm genau 25 Textzeilen, die maximale vertikale Anzahl also, darstellt und trotzdem am oberen Bildschirmrand die Seitennummer, sowie am unteren Bildschirmrand das MD-Logo zu sehen sind. Ganz offensichtlich haben wir es geschafft, den Bildschirm auf wundersame Weise zu vergrößern!
Tatsächlich schaltet das MD-Hauptprogramm den oberen und unteren Bildschirmrand schlichtweg ab, so daß wir auch hier noch etwas darstellen können und auf diese Weise mehr Informationen auf einer Bildschirmseite abzulesen sind. Dies ist wieder einmal nichts anderes als ein Rastertrick. Noch dazu einer der simpelsten die es gibt. Wie einfach er zu programmieren ist soll folgendes kleines Rasterprogramm verdeutlichen. Sie finden es auf dieser MD unter dem Namen "BORDERDEMO" und müssen es wie immer mit ",8,1" laden und mittels "SYS4096" starten:

```
INIT:   SEI             ;IRQ sperren
        LDA #$7F        ;Timer-IRQ
        STA $DC0D       ; abschalten
        LDA $ DC0D      ;ICR löschen
        LDA #$F8        ;Rasterzeile $F8 als
        STA $D012;      Interrupt-Auslöser
        LDA $D011       ;festlegen (incl. dem
        AND #$7F        ;Löschen des HI-Bits)
        STA $D011
        LDA #$01        ;Raster als IRQ-
        STA $D01A       ;Quelle wählen
        LDX #<(IRQ)     ;IRQ-Vektoren auf
        LDY #>(IRQ)     ;eigene Routine
        STX $0314       ;verbiegen
        STY $0315
        LDA #$00        ;Letzte VIC-Adr. auf
        STA $3FFF       ;0 setzen
        LDA #$0E        ;Rahmen- und Hinter-
        STA $D020       ;grundfarben auf
        LDA #$06        ;hellblau/blau
        STA $D021       ;setzen
        LDY #$3F        ;Sprite-Block 13
        LDA #$FF        ;($0340) mit $FF
LOOP2   STA $0340,Y     ;füllen
        DEY
        BPL LOOP2
```

```
                LDA #$01        ;Sprite 0
                STA $D015       ;einschalten
                STA $D027       ;Farbe="Weiß"
                LDA #$0D        ;Spritezeiger auf
                STA $07F8       ;Block 13 setzen
                LDA #$64        ;X- und Y-Pos.
                STA $D000       ;auf 100/100
                STA $D001       ;setzen
                CLI             ;IRQs freigeben
                RTS             ;ENDE
IRQ             LDA $D011       ;Bildschirm
                AND #$F7        ;auf 24 Zeilen
                STA $D011       ;Umschalten
                DEC $D019       ;VIC-ICR löschen
                LDX #$C0        ;Verzögerungs-
LOOP1           INX             ;schleife
                BNE LOOP1
                LDA $D011       ;Bildschirm
                ORA #$08        ;auf 25 Zeilen
                STA $D011       ;zurückschalten
                INC $D001       ;Sprite bewegen
END             JMP$EA31        ;Weiter zum SYS-IRQ
```

Wie Sie sehen, besteht die eigentliche IRQ-Routine, die den Border abschaltet nur aus einer handvoll Befehlen. Die Initialisierung der Routine sollte Ihnen noch aus dem letzten Kursteil bekannt sein. Wir sperren hier zunächst alle IRQs und verhindern, daß CIA-A ebenfalls IRQs auslöst, damit unser Rasterinterrupt nicht gestört wird. Als nächstes wird Rasterzeile $F8 als Interruptauslöser festgelegt, was auch einen ganz bestimmten Grund hat, wie wir weiter unten sehen werden. Nun sagen wir noch dem VIC, daß er Rasterinterrupts auslösen soll, und verbiegen den IRQ-Vektor bei $0314/$0315 auf unsere eigene Routine namens "IRQ". Die nun folgenden Zeilen dienen lediglich " kosmetischen" Zwecken. Wir setzen hier Rahmen und Hintergrundfarben auf die Standardwerte und schalten Sprite 0 ein, das von unserer Interruptroutine in der Vertikalen pro IRQ um einen Pixel weiterbewegt werden soll. Zudem wird der Spriteblock, der dieses Sprite darstellt, mit $FF gefüllt, damit wir ein schönes Quadrat auf dem Bildschirm sehen und keinen Datenmüll. Nach Freigabe der IRQs mittels "CLI" wird dann wieder aus dem Programm zurückgekehrt. Von nun an arbeitet unsere kleine, aber feine Raster-IRQ- Routine. Damit Sie sie verstehen, müssen wir nun ein wenig in die Funktionsweise des VIC einsteigen: Normalerweise zeigt uns der Videochip des C64, wie oben schon erwähnt, ein 25 Text-, bzw. 200 Grafikzeilen hohes Bild.
Nun können wir die Bildhöhe mit Hilfe von Bit 3 in Register 17 des VICs auf 24 Textzeilen reduzieren. Setzen wir es auf "1", so werden 25 Textzeilen dargestellt, setzen wir es auf "0", so sehen wir lediglich 24 Textzeilen. Im letzteren Fall werden dann jeweils vier Grafikzeilen des oberen und unteren Bildschirmrandes vom Bildschirmrahmen überdeckt. Diese Einschränkung ist vor allem bei der Programmierung eines vertikalen Soft-Scrollers von Bedeutung.
Effektiv zeichnet der VIC nun also den oberen Bildschirmrand vier Rasterzeilen länger und den unteren vier Rasterzeilen früher. Um nun den Rahmen zu zeichnen kennt die Schaltlogik des VIC zwei Rasterzeilen, die er besonders behandeln muß. Erreicht er nämlich Rasterzeile $F7, ab der der Bildschirm endet, wenn die 24 Textzeilen-Darstellung aktiv ist, so prüft er, ob Bit 3 von Register 17 gelöscht ist. Wenn ja, so beginnt er den Rand zu zeichnen, wenn nein, so fährt er normal fort. Erreicht er dann Rasterzeile $FB, die das Ende eines 25-zeiligen Bildschirms darstellt, wird nochmals geprüft, ob das obige Bit auf 0 ist. Wenn ja, so weiß der VIC, daß er mit dem Zeichnen des Rahmens schon begonnen hat. Wenn nein, so beginnt er erst jetzt damit. Mit unserem Interrupt tricksen wir den

armen Siliziumchip nun aus. Unsere Routine wird immer in Rasterzeile $F8 angesprungen, also genau dann, wenn der VIC die 24- Zeilen-Prüfung schon vorgenommen hat. Da die Darstellung auf 25 Zeilen war, hat er noch keinen Rand gezeichnet. Unsere Interruptroutine schaltet nun aber auf 24 Zeilen um und gaukelt dem VIC auf diese Weise vor, er hätte schon mit dem Zeichnen des Randes begonnen, weshalb er nicht noch einmal beginnen muß, und somit ohne zu zeichnen weitermacht. Dadurch erscheinen unterer und oberer Bildschirmrand in der Hintergrundfarbe, und es ist kein Rahmen mehr sichtbar. In diesen Bereichen kann man nun zwar keinen Text oder Grafik darstellen, jedoch sind Sprites, die sich hier befinden durchaus sichtbar! Sie werden normalerweise nämlich einfach vom Rahmen überdeckt, sind aber dennoch vorhanden. Da der Rahmen nun aber weg ist, sieht man auch die Sprites, wie das sich bewegende Sprite 0 unserer Interruptroutine beweist!
Wichtig an unserer Routine ist nun noch, daß wir vor Erreichen des oberen Bildrandes die Darstellung noch einmal auf 25 Zeilen zurückschalten, damit der Trick beim nächsten Rasterdurchlauf noch einmal klappt. Hierbei darf natürlich frühestens dann umgeschaltet werden, wenn der Rasterstrahl an der zweiten Prüf-Position, Rasterzeile $FB, schon vorbei ist.
Dies wird durch die kleine Verzögerungsschleife bewirkt, die genau 4 Rasterzeilen wartet, bevor mit dem anschließenden ORA-Befehl Bit 3 in Register 17 des VIC wieder gesetzt wird. Am Ende unseres Interrupts bewegen wir das Sprite noch um eine Y-Position weiter und verzweigen zum Betriebssystem-IRQ, damit die Systemaufgaben trotz abgeschalteter CIA dennoch ausgeführt werden. Die interruptauslösende Rasterzeile muß nicht nochmal neu eingestellt werden, da wir diesmal nur eine Rasterzeile haben, die jedesmal wenn sie erreicht wird einen Interrupt auslöst.
Wollen wir nun noch klären, warum wir bei der Initialisierung eine 0 in Adresse $3FFF geschrieben haben. Wie Sie vielleicht wissen, kann der VIC Speicherbereiche von lediglich 16 KB adressieren, aus denen er sich seine Daten holt. Im Normalfall ist das der Bereich von $0000-$3FFF. Die letzte Speicherzelle seines Adressbereichs hat nun eine besondere Funktion. Der Bit-Wert, der in ihr steht, wird nämlich in allen Spalten der Zeilen des nun nicht mehr überdeckten Bildschirmrandes dargestellt - und zwar immer in schwarzer Farbe. Durch das Setzen dieser Zelle auf 0 ist hier also gar nichts sichtbar. Schreiben wir jedoch bei aktivierter Borderroutine mittels " POKE16383, X" andere Werte hinein, so werden je nach Wert mehr oder weniger dicke, vertikale Linien in diesem Bereich sichtbar. Durch Setzen aller Bits mit Hilfe des Wertes 255(mit Rahmenfarbe= schwarz), können wir sogar einen scheinbar vorhandenen Bildschirmrand simulieren!
Vielleicht fällt Ihnen nun auch noch ein interessanter Nebeneffekt auf: nachdem wir die oberen und unteren Bildschirmgrenzen abgeschaltet haben, gibt es Spritepositionen, an denen das Sprite zweimal zu sehen ist. Nämlich sowohl im oberen, als auch im unteren Teil des Bildschirms. Das liegt daran, daß das PAL-Signal, welches der VIC erzeugt 313 Rasterzeilen kennt, wir aber die Y-Position eines Sprites nur mit 256 verschiedenen Werten angeben können.
Dadurch stellt der VIC das Sprite an den Y-Positionen zwischen 0 und 30 sowohl am unteren, als auch am oberen Rand dar.
Bei eingeschalteten Rändern fiel dieser Nebeneffekt nie auf, da diese Spritepositionen normalerweise im unsichtbaren Bereich des Bildschirms liegen, wo sie vom Bildschirmrahmen überdeckt werden.
Bleibt noch zu erwähnen, daß wir mit einem ähnlichen Trick auch die seitlichen Ränder des Bildschirms verschwinden lassen können, nur ist das hier viel schwieriger, da es auf ein sehr genaues Timing ankommt. Wie man damit umgeht müssen wir jetzt erst noch lernen, jedoch werde ich in den nächsten Kursteilen auf dieses Problem noch einmal zu sprechen kommen.

2. Einzeilen-Raster-Effekte

Kommen wir nun zu dem oben schon erwähnten Timing-Problem. Vielleicht haben Sie nach Studieren des letzten Kursteils einmal versucht einzeilige Farbrastereffekte zu programmieren. Das heißt also daß Sie gerade eine Zeile lang, die Bildschirmfarben wechseln, und sie dann wieder auf die Normalfarbe schalten.
Hierzu wären dann zwei Raster-Interrupts notwendig, die genau aufeinander zu folgen haben (z.B. in Zeile $50 und $51). Wenn Sie versucht haben ein solches Rasterprogramm zu schreiben, so werden Sie bestimmt eine Menge Probleme dabei gehabt haben, da die Farben nie genau eine Rasterzeile lang den gewünschten Wert enthielten, sondern mindestens eineinhalb Zeilen lang sichtbar waren. Dieses Problem hat mehrere Ursachen, die hauptsächlich durch die extrem schnelle Geschwindigkeit des Rasterstahls entstehen. Selbiger bewegt sich nämlich in genau 63 Taktzyklen einmal von links nach rechts.
Da innerhalb von 63 Taktzyklen nicht allzu viele Instruktionen vom Prozessor ausgeführt werden können, kann jeder Befehl zuviel eine zeitliche Verzögerung verursachen, die eine Farbänderung um mehrere Pixel nach rechts verschiebt, so daß die Farbe nicht am Anfang der Zeile, sondern erst in ihrer Mitte sichtbar wird. Da ein IRQ nun aber verhältnismäßig viel Rechenzeit benötigt, bis er abgearbeitet ist, tritt ein Raster-IRQ in der nächsten Zeile meist zu früh auf, nämlich noch bevor der erste IRQ beendet wurde! Dadurch gerät das Programm natürlich vollkommen aus dem Takt und kann ggf. sogar abstürzen!
Noch dazu muß ein weiterer, hardwaremäßiger Umstand beachtet werden: hat man normale Textdarstellung auf dem Bildschirm eingeschaltet, so muß der VIC nämlich jedes mal zu Beginn einer Charakterzeile die 40 Zeichen aus dem Video-RAM lesen, die er in den folgenden acht Rasterzeilen darzustellen hat, und sie entsprechend in ein Videosignal umwandeln. Um diesen Vorgang durchzuführen hält er den Prozessor für eine Zeit von genau 42 Taktzyklen an, damit er einen ungestörten Speicherzugriff machen kann.
Eine Charakterzeile ist übrigens eine der 25 Textzeilen. Da der Bildschirm in der Regel bei Rasterzeile $32 beginnt, und jede achte Rasterzeile ein solcher Zugriff durchgeführt werden muß, sind all diese Zeilen besonders schwierig über einen Raster-IRQ programmierbar, da erst nach dem VIC-Zugriff ein Raster-IRQ bearbeitet werden kann, der jedoch durch den Zugriff schon viel zu spät eintritt, da die Zeile in der Zwischenzeit ja schon zu zwei Dritteln aufgebaut wurde.
Hier muß man sich eines speziellen Tricks behelfen. Um selbigen besser erläutern zu können, wollen wir uns das folgende Beispielprogramm einmal etwas näher anschauen:

```
INIT    SEI             ;IRQs sperren
        LDA #$7F        ;CIA-A-IRQs
        STA $DC0D       ;unterbinden
        LDA $DC0D
        LDA #$82        ;Rasterzeile $82 als
        STA $D012       ;Interruptauslöser
        LDA $D011       ;festlegen (incl.
        AND #$7F        ;Löschen des
        STA $D011       ;HI-Bits
        LDA #$01        ;Rasterstrahl ist
        STA $D01A       ;IRQ-Auslöser
        LDX #<(IRQ)     ;IRQ-Vektor auf eigene
        LDY #>(IRQ)     ;Routine verbiegen
        STX $0314
        STY $0315
        CLI             ;IRQs freigeben
VERZ    RTS             ;ENDE

IRQ     DEC $D019       ;VIC-IRQ freigeben
        JSR VERZ        ;Verzögern...
```

```
                JSR VERZ
                NOP
                LDY #$00        ;Farb-Index init.
LOOP1           LDX #$08        ;Char-Index init.
LOOP2           LDA $1100,Y     ;Farbe holen
                STA $D020       ;und im Rahmen-und
                STA $D021       ;Hintergrund setzen
                INY             ;Farb-Index+1
                DEX             ;Char-Index-1
                BEQ LOOP1       ;Wenn Char=0 verzweig.
                LDA VERZ        ;Sonst verzögern...
                JSR VERZ
                JSR VERZ
                JSR VERZ
                CPY #$48        ;Farb-Index am Ende?
                BCC LOOP2       ;Nein also weiter
                LDA #$0E        ;Sonst Rahmen/Hinter-
                STA $D020       ;grund auf
                LDA #$06        ;Standardfarben
                STA $D021       ;zurücksetzen
                JMP $EA31       ;IRQ mit SYS-IRQ beenden
```

Besten laden Sie das Programm einmal und starten es mit "SYS4096". Sie sehen nun einen schönen Rasterfarbeneffekt auf dem Bildschirm, wo wir ab Rasterzeile $83 in jeder Zeile die Rahmen und Hintergrundfarbe ändern. Ich habe hierbei Farbabstufungen benutzt, die schöne Balkeneffekte erzeugen. Die entsprechende Farbtabelle liegt ab Adresse $1100 im Speicher und kann natürlich von Ihnen auch verändert werden.
Kommen wir nun zur Programmbeschreibung. Die Initialisierungsroutine sollte Ihnen keine Probleme bereiten. Wir schalten wie immer die CIA ab, sagen dem VIC, daß er einen Raster-IRQ generieren soll, legen eine Rasterzeile (hier Zeile $82) als IRQ-Auslöser fest, und verbiegen die IRQ-Vektoren auf unser eigenes Programm.
Etwas seltsam mag Ihnen nun die eigentliche IRQ-Routine vorkommen. Nachdem wir mit dem DEC-Befehl dem VIC bestätigt haben, daß der IRQ bei uns angekommen ist, folgen nun drei, scheinbar sinnlose, Befehle. Wir springen nämlich das Unterprogramm "VERZ" an, das lediglich aus einem RTS-Befehl besteht, und somit direkt zu unserem Programm zurück verzweigt. Zusätzlich dazu folgt noch ein NOP-Befehl der ebenso wenig tut, wie die beiden JSRs zuvor. Der Sinn dieser Instruktionen liegt lediglich in einer Zeitverzögerung, mit der wir abwarten, bis der Rasterstrahl am Ende der Zeile $82 angelangt ist. Wir hätten hier auch jeden anderen Befehl verwenden können, jedoch ist es mit JSRs am einfachsten zu verzögern, da ein solcher Befehl, ebenso wie der folgende RTS-Befehl, jeweils 6 Taktzyklen verbraucht.
Durch einen JSR-Befehl vergehen also genau 12 Taktzyklen, bis der nächste Befehl abgearbeitet wird. Da ein NOP-Befehl, obwohl er nichts macht, zwei Taktzyklen zur Bearbeitung benötigt, und wir zwei JSRs verwenden, verzögern wir also um insgesamt 26 Taktzyklen. Genau diese Verzögerung ist dem DEC-Befehl zuvor und den folgenden LDX und LDY- Befehlen notwendig, um soviel zu verzögern, daß sich der Rasterstrahl bis ans Ende der Rasterzeile bewegt hat.
Hinzu kommt daß wir die 42 Taktzyklen hinzurechnen müssen, die der VIC den Prozessor sowieso schon gestoppt hat, da Rasterzeile $82 eine der schon oben angesprochenen Charakterzeilen darstellt ($82-$32=$50/8=10- ohne Rest!).

Ich hoffe, Sie nun nicht unnötig mit dem Gerede von Taktzyklenzahlen verwirrt zu haben. Im Endeffekt kommt es darauf an, das Ende der entsprechenden Rasterzeile abgewartet zu haben. Wie viel Verzögerung dazu notwendig ist, muß nicht groß berechnet werden, sondern wird in der Regel einfach ausprobiert. Sie fügen der IRQ-Routine einfach so viele

Verzögerungen hinzu, bis eine Farbänderung genau in einer Zeile liegt, und nicht irgendwo mitten in der Rasterzeile anfängt.
Beachten Sie bitte, daß Sie die Verzögerung für eine Nicht-Charakterzeile erweitern müssen, da in diesen Zeilen dem Prozessor ja 42 zusätzliche Taktzyklen zur Verfügung stehen!
Kommen wir nun zu den folgenden Instruktionen. Auch hier haben wir es nicht einfach mit irgendeinem Programm zu tun, sondern mit einer sorgfältigen Folge von Befehlen, die genau darauf abgestimmt ist, immer solange zu dauern, bis genau eine Rasterzeile beendet ist. Wie ich zuvor erwähnte sind das immer genau 63 Taktzyklen pro Rasterzeile, in denen der Prozessor irgendwie beschäftigt sein muß, damit die nächste Farbänderung zum richtigen Zeitpunkt eintritt. Wie immer funkt uns jede achte Rasterzeile der VIC dazwischen, der den Prozessor dann wieder für 42 Takte anhält, weswegen unsere Routine jede achte Rasterzeile nicht mehr und nicht weniger als 63-42=21 Taktzyklen dauern darf! Da die nun folgende Beschreibung etwas haarig wird, und schnell in arithmetisches Taktzyklenjonglieren ausartet, hier noch einmal die Farbänderungsschleife aus unserem Beispielprogramm, wobei ich hier die Kommentare durch die Zyklenzahlen je Befehl ersetzt habe:

```
                LDY #$00        ;2
LOOP1           LDX #$08        ;2
LOOP2           LDA $1100,Y     ;4
                STA $D020       ;4
                STA $D021       ;4
                INY             ;2
                DEX             ;2
                BEQ LOOP1       ;2 oder 3
                LDA VERZ        ;4
                JSR VERZ        ;12
                JSR VERZ        ;12
                JSR VERZ        ;12
                CPY #$48        ;2
                BCC LOOP2       ;2 oder 3
```

Der LDY-Befehl am Anfang ist eigentlich weniger wichtig, ich habe ihn nur der Vollständigkeit halber aufgeführt. Wir haben hier zwei verschachtelte Schleifen vor uns. Die eine, mit dem Namen "LOOP1" wird immer nur jede achte Rasterzeile aufgerufen, nämlich dann, wenn eine Charakterzeile beginnt. Diese Schleife wird über das X-Register indiziert. Die zweite Schleife wird vom Y-Register gesteuert, das gleichzeitig Indexregister für unsere Farbtabelle bei $1100 ist.
Wichtig ist nun der zeitliche Ablauf der beiden Schleifen. Wie wir ja wissen, müssen wir in einer Charakterzeile mit unserem Programm 21 und in einer normalen Rasterzeile 63 Taktzyklen verbrauchen. Da wir uns beim ersten Schleifendurchlauf genau in Rasterzeile $83 befinden, beginnt die Schleife also zunächst in einer normalen Rasterzeile (eine Zeile nach einer Charakterzeile).
Hier wird die Schleife ab dem Label "LOOP2" bis zum Ende ("BCC LOOP2") abgearbeitet. Wenn Sie jetzt die Taktzyklen am Rand innerhalb dieses Bereichs aufaddieren, so vergehen bis zum BCC-Befehl genau 60 Zyklen. Der BCC-Befehl hat nun eine ganz besondere Funktion. Alle Branch-Befehle verbrauchen nämlich bei nicht zutreffender Abfragebedingung nur zwei Taktzyklen (so auch beim zuvorigen BEQ-Befehl der das X-Register abfragt).
Trifft die Bedingung zu, so wie auch beim abschließenden BCC, so muß verzweigt werden, was einen weiteren, dritten Taktzyklus in Anspruch nimmt.
Dadurch sind also genau 60+3=63 Taktzyklen verstrichen, wenn die Schleife das nächste Mal durchlaufen wird. Und das ist genau die Zeit die vergehen muß, bis der Rasterstrahl in

der nächsten Zeile ist, wo die Farbe erneut geändert werden kann. Kommt der Strahl nun wieder in eine Charakterzeile, so ist das X-Register auf Null heruntergezählt. Durch die zutreffende Abfragebedingung im BEQ-Befehl dauert die Verzweigung nun drei Takte. Vom Label "LOOP2" bis zu dem BEQ-Befehl verbrauchen wir also nach Adam Riese nun 19 Taktzyklen. Da der Branch-Befehl zum Label "LOOP1" verzweigt, und der dortige LDX-Befehl wiederum 2 Zyklen benötigt, sind genau 21 Takte verstrichen, wenn sich der Prozessor wieder am Startpunkt,"LOOP2" nämlich, befindet.
Und das ist wieder genau die Zeit die verstreichen musste, damit in der Charakterzeile der Rasterstrahl wieder am Anfang der folgenden Zeile steht! Sie sehen also, wie sehr es auf genaues Timing hier ankommt! Fügen Sie dieser Kehrschleife auch nur einen Befehl hinzu, oder entfernen Sie einen, so gerät das gesamte Timing außer Kontrolle und unsere Farbbalken erscheinen verzerrt auf dem Bildschirm. Probieren Sie es ruhig einmal aus!
Zum Abschluß des Raster-IRQs schalten wir nun wieder die normalen Bildschirmfarben ein und verzweigen zum Betriebssystems-IRQ.

3. Weitere Programmbeispiele
Außer den beiden bisher besprochenen Programmen finden Sie auf dieser MD noch drei weitere Beispiele, die lediglich Variationen des letzten Programms darstellen. Alle drei werden wie immer mit ",8,1" geladen und mit "SYS4096" gestartet. "RASTCOLOR2" entspricht haargenau "RASTCOLOR1", nur daß ich hier am Ende eine Routine hinzugefügt habe, die die Farbtabelle um jeweils eine Zeile weiterrotiert. Das Ergebnis des Ganzen sind rollende und nicht stehende Farbbalken.
Die Programme "RASTSINUS1" und "-2" funktionieren nach einem ähnlichen Prinzip. Hier wird jedoch nicht die Farbe in den angegebenen Rasterzeilen verändert, sondern der horizontale Verschiebeoffset. Dadurch kann der entsprechende Bildbereich effektvoll verzerrt werden. Starten Sie "RASTSINUS1" und fahren Sie mit dem Cursor in die untere Bildschirmhälfte, so werden dort alle Buchstaben in Form einer Sinuskurve verzerrt. "RASTSINUS2" geht noch einen Schritt weiter. Hier werden die Werte der Sinustabelle, wie auch schon bei "RASTCOLOR2" am Ende der Interruptroutine gerollt, weswegen der gerasterte Bereich, wasserwellenähnlich hin und her "schlabbert". Schauen Sie sich die Programme ruhig einmal mit Hilfe eines Speichermonitors an, und versuchen Sie ein paar Änderungen daran vorzunehmen. Im nächsten Kursteil werden wir noch ein wenig mehr mit Taktzyklen herumjonglieren und uns mit FLD und Sideborder-Routinen beschäftigen.

(ub)

Teil 5 – Magic Disk 03/94

Nachdem wir im letzten Monat ja schon kräftig mit schillernden Farb- und Sinuswellenrasterroutinen um uns geworfen haben, möchten wir uns auch in dieser Ausgabe der MD einem sehr trickreichen Beispiel eines Raster-IRQs zuwenden: der FLD-Routine.

1. FLD - ein Zauberwort für Rasterfreaks
Die Abkürzung "FLD" steht für "Flexible Line Distance", was übersetzt soviel bedeutet wie "beliebig verschiebbarer Zeilenunterschied". Diese, zunächst vielleicht etwas verwirrende, Bezeichnung steht für einen Rastereffekt, der vom Prinzip und der Programmierung her extrem simpel ist, jedoch ungeahnte Möglichkeiten in sich birgt. Um zu wissen, welcher Effekt damit gemeint ist, brauchen Sie sich lediglich einmal anzuschauen, was passiert, wenn Sie im MD-Hauptmenü einen neuen Text laden, oder einen gelesenen Text wieder verlassen:

der Textbildschirm scheint hier von unten her hochgezogen, bzw. nach unten hin weggedrückt zu werden. Und genau das tut nun eine FLD-Routine. Hierbei sei darauf hingewiesen, daß es sich dabei nicht um irgendeine Programmierakrobatik handelt, bei der aufwendig hin- und herkopiert und rumgescrollt werden muß, sondern um eine einfache, ca.150 Byte große, Rasterroutine! Der Trick des Ganzen liegt wie so oft bei der Hardware des 64ers, die wieder einmal beispielhaft von uns "veräppelt" wird. Denn eigentlich sollte sie nicht dazu in der Lage sein, einen solchen Effekt zu erzeugen!
Wie funktioniert nun diese Routine? Wie Sie vielleicht wissen, kann in den unteren drei Bits von Register 17 des VICs ($D011), ein vertikaler Verschiebeoffset für die Bildschirmdarstellung eingetragen werden. In der Regel benutzt man diese Bits um ein vertikales Softscrolling zu realisieren. Je nach dem welcher Wert dort eingetragen wird (von 0 bis 7), kann die Darstellung des sichtbaren Bildschirms um 0 bis 7 Rasterzeilen nach unten verschoben werden. Lässt man diese Werte nacheinander durch das Register laufen, und kopiert man daraufhin den Inhalt des Bildschirms von der 2. Textzeile in die 1. Textzeile, so entsteht ein Softscrolling nach unten. Der Wert, der dabei in den unteren drei Bits von $D011 steht gibt dem VIC an, ab welchem vertikalen Bildschirmoffset er damit anfangen soll, die nächste Textzeile aufzubauen. Wie wir aus dem letzten Kursteil noch wissen, geschieht dies ab Rasterzeile 41 und jeweils in jeder achten, folgenden Zeile. Wird nun ein vertikaler Verschiebeoffset angegeben, so verzögert der VIC diesen Zeitpunkt um die angegebene Anzahl Rasterzeilen (maximal 7). Steht in der Vertikalverschiebung z.B. der Wert 1, so muß der VIC also noch eine Rasterzeile warten, bis er die nächste Charakterzeile aufzubauen hat.
Der Trick der FLD-Routine liegt nun darin, daß Sie in jeder Rasterzeile, diesen Charakterzeilenanfang vor dem Rasterstrahl "herschiebt", so daß dieser eigentlich nie die gesuchte Anfangszeile erreichen kann - zumindest nicht solange, wie unsere FLD-Routine ihm vortäuscht, noch nicht den Anfang dieser Zeile erreicht zu haben! Wie einfach das alles geht, soll Ihnen folgendes Programmbeispiel verdeutlichen. Sie finden es auf dieser MD unter dem Namen "FLD-DEMO1" und müssen es absolut (mit ",8,1") laden und wie alle unsere Programmbeispiele mit "SYS4096" starten:

```
init:   SEI             ;IRQs sperren
        LDA #$7F        ;CIA-Timer abschalten
        STA $DC0D       ;(SYS-IRQ)
        LDA $DC0D       ;und CIA-ICR löschen
        LDA #$F8        ;Zeile $f8 ist IRQ-
        STA $D012       ; Auslöser
        LDA $D011       ;7.Bit Rasterzeile
        AND #$7F        ;löschen
        STA $d011       ;und     zurückschreiben
        LDA #$01        ;VIC löst Raster-IRQs
        STA $D01A       ;aus
        LDX #<(IRQ2)    ;IRQ-Vektor auf
        LDY #>(IRQ2)    ;eigene Routine
        STX $0314       ;verbiegen
        STY $0315
        LDA #$00        ;Zählregister
        STA $02         ;löschen
        LDA #$FF        ;Leerbereich auf
        STA $3FFF       ;schwarz setzen
        CLI             ;IRQs freigeben
verz:   RTS             ;und Ende

irq1:   LDA #$10        ;Vert. Verschiebung
        STA $D011       ;gleich 0
        LDA $02         ;Zähler laden
        BEQ lab2        ;wenn 0, überspringen
```

```
                LDX #$00        ;Zählregister löschen
        lab1:   CLC             ;Carry f.Add.löschen
(!)             LDA $D011       ;Verschiebung holen
(!)             ADC #$01        ;+1
(!)             AND #$07        ;untere Bits ausmaskieren
(!)             ORA #$10        ; Bit 4 setzen
(!)             STA $D011       ;und zurückschreiben.
                DEC $D019       ;VIC-IRQ freigeben
                JSR verz        ;Verzögern
                JSR verz
                LDA $d012       ;Strahlpos < Bild-
                CMP #$F6        ;schirmende?
                BEQ lab2        ;Ja, also überspringen
                INX             ;Zähler+1
                CPX $0002       ;Zähler=Endwert?
                BNE lab1        ;Nein, also weiter
        lab2:   LDA #$F8        ;Rasterzeile $f8 ist
                STA $D012       ;nächster IRQ-Ausl.
                DEC $D019       ;VIC-IRQs freigeb.
                LDA #$78        ;IRQ-Vektor auf IRQ2-
                STA $0314       ;Routine verbiegen
                LDX #$0E        ;Bildschirmfarben
                LDY #$06        ;zurücksetzen
                STX $D020
                STY $D021
                JMP $FEBC       ;IRQ beenden

        irq2:   LDA #$10        ;Vertikal-Versch.
                STA $D011       ; init.
                LDA #$71        ;Rasterz. $71 ist
                STA $D012       ;IRQ-Auslöser
                DEC $D019       ;VIC-IRQs freigeben
                LDA #$30        ;IRQ-Vektor auf IRQ1-
                STA $0314       ;routine verbiegen
                LDA $DC00       ;Portregister lesen
                LSR             ;Akku in Carry rotieren
                BCS lab3        ;C=1? Wenn ja, weiter
                DEC $02         ;sonst Joystick hoch
        lab3:   LSR             ;Akku in Carry rotieren
                BCS lab4        ;C=1? Ja, also weiter
                INC $02         ;Sonst Joyst. runter
        lab4:   JMP $EA31       ;SYS-IRQ und Ende
```

Die Beschreibung der Initialisierungsroutine können wir uns sparen, da wir ihren Aufbau ja schon von anderen Programmbeispielen her kennen. Wichtig ist nur, daß wir hier Rasterzeile $F8 als IRQ-Auslöser festlegen, und die zweite IRQ-Routine ("IRQ2") in den IRQ-Vektor eintragen. Ansonsten wird hier auch noch der FLD-Zeilenzähler in Speicherzelle $02 gelöscht, sowie der Wert $FF in letzte Adresse des VIC-Bereichs geschrieben. Die Bedeutung dieser Adresse kennen wir noch von unserer Borderroutine aus dem letzten Kursteil: ihr Inhalt wird in schwarzer Farbe immer an allen Stellen auf dem Bildschirm angezeigt, an denen wir den Rasterstrahl mit unseren Interrupts austricksen, was ja auch hier der Fall ist. Möchten wir an solchen Stellen die Hintergrundfarbe sehen, so müssen wir den Wert $00 hineinschreiben.
Die Routine "IRQ2" wird nun immer einmal pro Bildschirmaufbau aufgerufen. Sie bereitet die eigentliche FLD-Routine vor, die ab der Rasterzeile $71 ausgelöst werden soll. Gleichzeitig beinhaltet diese Routine eine Joystickabfrage, mit der wir das Zählregister in Adresse $02 ändern können. Auf diese Weise kann mit dem Joystick die FLD-Lücke ab Rasterzeile $71, je nach Wunsch, vergrößert oder verkleinert werden. Abschließend biegt

diese IRQ-Routine den IRQ-Vektor auf die eigentliche FLD-IRQ- Routine ("IRQ1") und ruft den System-IRQ auf, den wir in der Init-Routine ja abgeschaltet hatten und nun "von Hand" ausführen müssen.
Hiernach ist nun "IRQ1" am Zug. Kern der Routine ist die Schleife zwischen den beiden Labels "LAB1" und "LAB2" . Am wichtigsten sind hierbei die fünf Befehle die ich Ihnen mit Ausrufungszeichen markiert habe. Hier wird zunächst der Inhalt des Registers $D011 gelesen, in dem der vertikale Verschiebeoffset zu finden ist, und 1 auf diesen Wert hinzuaddiert. Da dabei auch ein Überlauf in das 3. Bit des Registers stattfinden kann, das ja nicht mehr zur vertikalen Verschiebung herangezogen wird, müssen wir mit dem folgenden AND-Befehl alle Bits außer den unteren dreien ausmaskieren, und mittels ORA, das 3. Bit wieder setzen, da es steuert, ob der Bildschirm ein, oder ausgeschaltet sein soll, und deshalb immer gesetzt sein muß. Anschließend wird der neue Wert für $D011 wieder zurückgeschrieben. Da diese Verschiebungsänderung nun auch in jeder folgenden Zeile auftreten soll, solange bis der Zeilenzähler abgelaufen ist, müssen mit dem Rest der Routine die 63 Taktzyklen, die der Rasterstrahl zum Aufbau einer Rasterzeile braucht, verzögert werden. Eine Unterscheidung in normale Rasterzeilen und Charakterzeilen, in denen der Prozessor vom VIC ja für 42 Taktzyklen angehalten wird, und die Schleife deshalb weniger verzögern muß, braucht diesmal nicht durchgeführt werden, da wir durch das "Vor uns Herschieben" der nächsten Charakterzeile deren Aufbau ja solange verhindern, bis die Schleife der FLD-Routine beendet ist. Dies ist dann der Fall, wenn entweder der Zähler im X-Register bis auf 0 gezählt wurde, oder aber die Rasterzeile $F6 erreicht wurde, ab der der untere Bildschirmrand beginnt.
Ab dem Label "LAB2", wird nun wieder Rasterzeile $F8 für "IRQ2" als Interruptauslöser festgelegt. Zusätzlich verbiegen wir den IRQ-Vektor auf diese Routine zurück. Dabei wird in unserem Beispiel lediglich das Low-Byte geändert, da beide Routinen ja an einer Adresse mit $10xx anfangen, und somit die High-Bytes der beiden Routinenadressen immer den Wert $10 haben. Zum Schluß wird wieder auf den Teil der Betriebssystemroutine ($FEBC) gesprungen, der die Prozessorregister vom Stack zurückholt und den Interrupt beendet.
Die Art und Weise, wie wir hier die Vertikalverschiebung vor dem Rasterstrahl herschieben mag etwas umständlich anmuten. Tatsächlich gibt es hier auch noch andere Möglichkeiten, die in den Beispielprogrammen "FLD-DEMO2", und "FLD-DEMO3" benutzt wurden. Sauberer ist die Lösung des Zeilenproblems, wenn man das Register, das die aktuelle Rasterzeile enthält ($D012), als Zähler verwendet.
Wir müssen hier lediglich die Rasterposition auslesen, ihren Wert um 1 erhöhen, die unteren drei Bits ausmaskieren und das 4. Bit in diesem Register wieder setzen. Selbiges wird durch die folgende Befehlsfolge durchgeführt:

```
CLC
LDA $D012
ADC #$01
AND #$07
ORA #$10
STA $D011
```

Noch schneller geht das, wenn man den illegalen Opcode "ORQ" verwendet. Er addiert 1 auf den Akku hinzu und verodert gleichzeitig das Ergebnis mit dem Operandenwert. Die Befehlsfolge ist dann nur noch vier Zeilen lang:

```
LDA $D012
AND #$07
ORQ #$10
STA $D011
```

Selbst wenn diese Methode kürzer ist, als die zuvor genannte, ist es dennoch nicht ratsam sie zu verwenden, da "ORQ" wie gesagt ein illegaler, also inoffizieller, Assemblerbefehl ist, und deshalb von den meisten Assemblern und Disassemblern nicht erkannt wird. Zudem können Laufzeitunterschiede oder gar Fehlfunktionen bei verschiedenen Produktionsversionen des 6510- Prozessors vorkommen, so daß ein Programm mit einem solchen illegalen Opcode nicht auf jedem C64 lauffähig sein muß. Wer es wirklich kurz will, der sollte über eine Tabelle die benötigten Zeilendaten holen, wie das im Beispiel "FLD-DEMO3" der Fall ist. Hier wurde eine Tabelle bei Adresse $1200 abgelegt, die den jeweils entsprechenden Wert für jede einzelne Rasterzeile enthält. Die eigentlichen FLD-Befehle verkürzen sich damit auf die beiden folgenden Zeilen:

```
LDA $1200,X
STA $D011
```

Die Lösung des Problems über eine Tabelle beinhaltet gleichzeitig auch noch den Vorteil, daß wir viel flexibler die FLD-Effekte einsetzen können. So ist es damit sehr einfach möglich, mehrere Charakterzeilen zu verschieben, wie das im "FLD-DEMO3" der Fall ist. Dieses Beispielprogramm beginnt übrigens ausnahmsweise an Adresse $1100, weswegen es nicht wie sonst mit "SYS4096", sondern durch ein "SYS4352" aufgerufen werden muß. Alles in allem sollten Sie sich die drei Beispiele ruhig einmal mit einem Disassembler oder Speichermonitor anschauen um ihre Funktionsweise zu verstehen. Mit FLD erzielbare Effekte sind sehr vielseitig und sie sollten schon ein wenig damit herumexperimentieren. Weiterhin gibt es einige Rastereffekte die durch eine FLD-Routine stark vereinfacht programmiert werden können, oder sogar ohne sie gar nicht möglich wären, weswegen ein gründliches Verständnis der Materie sehr von Vorteil bei anderen Rastereffekten sein kann.

2. Timingprobleme und Taktzyklenmesser

Wie wir wieder einmal bewiesen haben, ist die Rasterprogrammierung eine Sache, bei der es auf absolut exaktes Timing ankommt. Noch haariger wird das im nächsten Kursteil ersichtlich, wo wir Ihnen eine Sideborder-Routine vorstellen werden. Wird diese Routine auch nur einen Taktzyklus zu früh oder zu spät ausgeführt, so funktioniert sie schon nicht mehr. Deshalb wollen wir uns nun erst wieder ein wenig in die Theorie stürzen und Verfahrensweisen zur Ermittlung der Laufzeit eines Programms vorstellen.
Wie Sie mittlerweile nun oft genug mitbekommen haben, braucht der Rasterstrahl zum Aufbau einer Rasterzeile genau 63 Taktzyklen. Innerhalb dieser Zeit müssen wir den Prozessor immer irgendwie beschäftigen, damit wir rechtzeitig zum Beginn der nächsten Rasterzeile eine weitere Änderung vornehmen können. Hinzu kommt, daß wir bei eingeschaltetem Textmodus und Rastereffekten im sichtbaren Bildschirmbereich beachten müssen, daß jede achte Rasterzeile, jeweils am Beginn einer Charakterzeile, der VIC den Prozessor für 42 Taktzyklen anhält, damit er die, in den folgenden acht Rasterzeilen darzustellenden, Zeichen generieren kann. Somit bleiben für den Prozessor für solch eine Rasterzeile nur noch 21 Taktzyklen Rechenzeit. Um nun ein exaktes Timing zu erreichen müssten wir eigentlich die Laufzeiten eines je- den einzelnen Befehls einer Raster-Routine zusammenaddieren um herauszufinden, ob eine Routine schnell, bzw. langsam genug, abgearbeitet wird. Das kann unter Umständen eine sehr aufwendige Sache werden, da hierbei ewig lang Befehlstabellen mit Zyklenangaben gewälzt werden müssten, und bei jeder kleinen Änderung neue Verzögerungsbefehle in die Routine eingefügt, oder aus ihr entfernt werden müssten.
Damit Sie die Zyklenzahlen selbst zur Hand haben, habe ich Ihnen am Ende dieses Kurses in einer Tabelle alle Prozessor- Befehle in allen möglichen Adressierungsarten aufgelistet. Um also von Hand die Laufzeit einer Routine zu berechnen können Sie dort

nachschlagen.
Noch einfach geht das Abwägen der Laufzeit jedoch mit einem Programm. Wir können uns hier die Möglichkeit zunutze machen, daß mit den Timern der CIAs einzelne Zyklen gezählt werden können. Ich habe Ihnen hierzu ein Zyklenmessprogramm geschrieben, das es Ihnen ermöglicht, eine eigene Routine bezüglich ihrer Laufzeit zu testen. Es heißt "Cyclecount" und ist ebenfalls auf dieser MD zu finden. Das Programm ist in der Lage, Routinen mit einer Dauer von maximal 65490 Taktzyklen zu stoppen. Laden Sie es hierzu mit LOAD"CYCLECOUNT",8,1 in den Speicher und schreiben Sie Low- und High-Byte der zu testenden Routine in die Adressen 828/829 ($033C/$033D). Die zu messende Routine muß mit einem "BRK"- Befehl beendet werden. Rufen Sie nun das Programm mit einem "SYS49152" auf. Cyclecount gibt Ihnen daraufhin den ermittelten Zyklen-Wert auf dem Bildschirm aus. Das Programm benutzt dabei den Timer A von CIA-B zum Messen der Zyklen. Es initialisiert diesen Timer mit dem Wert $FFFF, startet ihn und ruft daraufhin die zu testende Routine auf.
Zuvor wird der BRK-Vektor bei Adresse $0316/$0317 auf eine eigene Routine verbogen. Wird die zu testende Routine nun mit einem BRK-Befehl beendet, so wird sofort zur Auswertungsroutine von Cyclecount verzweigt, die den Timer wieder anhält und den in den Timerregistern enthaltenen Wert von $FFFF subtrahiert.
Zudem müssen 45 Zyklen abgezogen werden, die hauptsächlich zur Ausführung des JMP-Befehls auf die zu testende Routine und durch den beendenden BRK-Befehl verbraucht wurden, und nicht mitgezählt werden dürfen.

3. Die Zyklentabelle

Kommen wir nun noch zu der Tabelle mit den Zyklendauern der einzelnen Befehle.
Da viele darunter mit unterschiedlichen Adressierungsarten verwendbar sind, habe ich Ihnen zwei Einzeltabellen aufgeführt, in denen ähnliche Befehle zu Gruppen zusammengefasst wurden. Kommen wir hierbei zunächst zu den impliziten Befehlen. Sie bestehen lediglich aus einem Befehlswort und benötigen keinen Operanden:

Befehl	**Zyklen**	**Befehl**	**Zyklen**
ASL	2	BRK	7
LSR	2	TAX	2
ROL	2	TAY	2
ROR	2	TXA	2
CLC	2	TYA	2
SEC	2	TXS	2
CLD	2	TSX	2
SED	2	PLA	4
CLI	2	PHA	3
SEI	2	PLP	4
CLV	2	PHP	3
NOP	2	INX	2
RTS	6	DEX	2
RTI	6	INY	2
		DEY	2

Es folgen nun die Befehle, die entweder direkt, oder über Speicheradressen eine Operation mit den Prozessorregistern durchführen. Die Bitschiebebefehle kommen hier

nochmals vor, da sie auch mit Adressierung verwendbar sind. Die Spalten der Tabelle stehen (von links nach rechts) für: "IMMediate = unmittelbar", wenn der Operand ein konstanter Wert ist ("LDA #00"), "ABSolut", für direkte Speicheradressierung ("LDA $1000"), "ABSolut,X" ("LDA $1000,X"), "ABSolut,Y", "ZeroPage" ("LDA $02"), "ZeroPage,X", "ZeroPage,Y", Zeropage indirektimplizit "(zp,X)" und Zeropage implizitidirekt "(zp),Y". Alle Zyklenangaben, die mit einem "*" markiert sind verlängern sich um einen Taktzyklus, wenn bei dieser Adressierung eine Bereichsüberschreitung stattfindet, was bedeutet, daß wenn die Summe des Offsetregisters und des Basiswertes das High-Byte überschreitet, ein Takt mehr benötigt wird, als angegeben. Dies ist z.B. bei dem Befehl "LDA $10FF,X" der Fall. Dann, wenn nämlich im X-Register ein Wert größer oder gleich 1 steht:

Befehl	**IMM**	**Abs**	**Abs ,X**	**Abs ,Y**	**ZP**	**ZP , X**	**ZP , Y**	**ZP (,X)**	**ZP (,Y)**
BIT		4			3				
CPX	2	4			3				
CPY	2	4			3				
CMP	2	4	4*	4*	3	4		6	5*
ADC	2	4	4*	4*	3	4		6	5*
SBC	2	4	4*	4*	3	4		6	5*
AND	2	4	4*	4*	3	4		6	5*
ORA	2	4	4*	4*	3	4		6	5*
EOR	2	4	4*	4*	3	4		6	5*
INC		6	7		5	6			
DEC		6	7		5	6			
LDA	2	4	4*	4*	3	4		6	5*
LDX	2	4	4*	3		4			
LDY	2	4	4		3	4*			
STA		4	5	5	3	4		6	6
STX		4			3		4		
STY		4			3	4			
ASL		6	7		5	6			
LSR		6	7		5	6			
ROL		6	7		5	6			
ROR		6	7		5	6			
JMP		3							
JSR		6							

Zudem kennt der JMP-Befehl auch noch die indirekte Adressierung, über einen Low-/ High-Bytezeiger (z.B."JMP ($A000)"), der 5 Taktzyklen verbraucht.
Es fehlen jetzt nur noch die Branch-Befehle, die jedoch nicht in einer eigenen Tabelle erscheinen müssen, da sie immer 2 Taktzyklen verbrauchen. Es sei denn, die vom Befehl abgefragte Bedingung trifft zu. In diesem Fall wird ein weiterer, dritter Takt in Anspruch genommen.
Das war es dann wieder für diesen Monat.
Im nächsten Kursteil werden wir uns weiterhin ein wenig mit Timingproblemen beschäftigen müssen, und uns ansehen, wie man IRQs "glättet". Dies soll uns dann als Grundlage für den nächsten Raster-Effekt dienen: einer Routine zum Abschalten der

seitlichen Ränder des Bildschirms.

(ub)

Teil 6 – Magic Disk 04/94

Anhand der FLD-Routine des letzten Kursteils hatten wir gesehen, wie einfach man die Hardware unseres kleinen Brotkastens austricksen kann, und sie dazu bewegt Dinge zu tun, zu denen sie eigentlich nicht in der Lage ist. So soll es auch in den folgenden Teilen des IRQ-Kurses sein, jedoch müssen wir uns zuvor um ein Problem kümmern, mit dessen Lösung wir noch trickreichere Rastereffekte programmieren und den Copper und Blittermagiern des Amigas das Fürchten lehren werden…

1. Auf gutes Timing kommt es an...
Wie auch schon am Ende des letzten Teils angesprochen, ist der Schlüssel zu tollen Rasterinterrupts ein besonders exaktes Timing. Wie schon am Beispiel der FLD-Routine unschwer erkennbar war, besteht das eigentliche "Austricksen" des Video-Chips meist aus gerade einer handvoll Befehlen. Wichtig ist nur, daß diese Befehle zum richtigen Zeitpunkt ausgeführt werden. Wie wichtig das ist, werden wir später am Beispiel einer Rasterroutine sehen, die in der Lage ist, den linken und rechten Rand des Bildschirms abzuschalten. Wird sie auch nur einen Taktzyklus zu früh oder zu spät ausgeführt, so bewirkt sie absolut gar nichts. Nur wenn zu einem ganz bestimmten Zeitpunkt der Wert eines Registers verändert wird, funktioniert sie auch wie sie soll! Damit wir solche Effekte also auch realisieren können, werden wir uns nun zwei Problemen widmen, die immer noch Ungenauigkeitsfaktoren in unseren Routinen darstellen und eliminiert werden müssen:

2. Systemvektoren sind schneller
Der erste Ungenauigkeitsfaktor ist das Betriebssystem des C64 . Wie Sie ja aus den ersten Kursteilen wissen, holt sich der 6510-Prozessor bei einer Interruptanfrage zunächst einmal eine der drei Interrupt-Sprungadressen am Ende seines Adressierungsbereichs (von $FFFA-$FFFF) in den Programmzähler. Hier stehen die jeweiligen Adressen der Betriebssystemsroutinen, die die entsprechende Art von Interrupt (IRQ, NMI oder Reset) bedienen. Gerade beim IRQ ist diese Routine etwas aufwendiger aufgebaut, da derselbe Vektor auch für softwaremäßige BRK-Interrupts benutzt wird, und die Routine deshalb eine Unterscheidung treffen muß.
Dadurch stiehlt sie uns quasi Prozessorzeit, wenn wir davon ausgehen, daß der BRK-Interrupt in der Regel nicht verwendet wird, da er ja einfacher durch ein JMP programmiert werden kann. Erst nach der Unterscheidung verzweigt die Routine dann über den Vektor $0314/$0315 auf die eigentliche IRQ-Routine (bzw. über $0316/$0317 zur BRK-Routine). Und erst an dieser Stelle "klinken" wir unsere eigenen Raster-IRQs in das Interruptsystem ein. Um nun die Verzögerung durch das Betriebssystem zu eliminieren, müssen wir es umgehen. Es sollte eigentlich also reichen, wenn wir die Startadresse unserer Interruptroutine im Vektor $FFFE/$FFFF eintragen, da er der IRQ-Vektor ist. Hierbei stellt sich uns jedoch ein weiteres Problem in den Weg:
diese Adressen gehören ja zum Betriebssystem-ROM und können nicht verändert werden, da sie für alle Zeiten in dem ROM-Chip eingebrannt sind. Aber nicht verzagen, denn "unter" diesem ROM befindet sich auch noch echtes RAM, und das können wir verändern. Damit der Prozessor sich dann den IRQ-Vektor auch von dort holt, müssen wir das darüberliegende ROM sozusagen "ausblenden", was über das Prozessoradressregister geschieht, das in Speicherzelle 1 zu finden ist. Im Normalfall steht hier der Wert 55 ($37), der das Basicund Betriebssystem-ROM in den Adressbereichen $A000-$BFFF (Basic), sowie $E000-$FFFF (System) einblendet.

Führen wir Schreibzugriffe auf diese Bereiche aus, so landen diese, selbst bei eingeschaltetem ROM im darunterlegenden RAM. Das Problem dabei ist nur, daß wir dann die geschriebenen Werte nicht auslesen können, da wir immer nur den Inhalt der ROM-Adresse lesen können.
Um das zu ändern, muß der Wert 53 ($35) in das Prozessoradressregister geschrieben werden. Dadurch werden nämlich die beiden ROM-Bausteine deaktiviert und das darunterliegende RAM kommt zum Vorschein, worauf der Prozessor dann auch Zugriff hat.
Ändern wir nun den IRQ-Vektor bei $FFFE/$FFFF, und tritt dann ein IRQ auf, so wird direkt auf unsere IRQ-Routine verzweigt, ohne daß der Umweg über das Betriebssystem gegangen wird. Beachten Sie hierbei jedoch, daß durch das Abschalten des ROMs weder Basic noch Betriebssystemroutinen verfügbar sind, da wir sie ja weggeschaltet haben. Benutzt Ihr eigenes Programm solche Routinen, so müssen Sie das ROM zuvor ins RAM kopieren. Dies tun Sie, indem Sie einfach bei eingeschaltetem ROM eine Adresse auslesen und gleich wieder in sie zurückschreiben. Beim Lesen erhalten Sie dann den Wert des ROMs, beim Schreiben schicken Sie ihn ins RAM darunter. Jetzt können Sie getrost das ROM abschalten, ohne daß Ihr Rechner abstürzt. Als Beispiel zum Kopieren der beiden ROMs ins darunterliegende RAM können Sie sich das Programm "COPYSYS" auf dieser MD anschauen, das wie all unsere Beispielprogramme absolut (mit ",8,1") geladen werden muß, und mit SYS4096 (JMP $1000) gestartet wird.
In unseren Beispielen werden wir allerdings keinerlei Betriebssystemsroutinen verwenden, weswegen wir uns hier das Kopieren einsparen. Wichtig ist dabei, daß wir die Interruptquellen von CIA-A sperren, damit sie uns mit ihren IRQs nicht zwischen die Raster-IRQs "funkt".

3. Das "Glätten" von Interrupts

Wenn Sie schon ein wenig mit Raster-IRQs "herumgespielt" haben, so wird Ihnen vielleicht schon einmal folgendes Problem aufgefallen sein: Möchten Sie einen Interrupt programmieren, der z. B. einfach nur die Hintergrundfarbe ändert, so passiert es manchmal, daß gerade an der Stelle, an der die Farbe geändert wird, ein unruhiges Flackern zu sehen ist. Man hat den Eindruck, als würde die Farbe mal ein paar Pixel früher oder später geändert werden, so daß an der selben Stelle des Bildschirms manchmal die alte, manchmal die neue Farbe zu sehen ist. Irgendwie scheint es also nicht möglich zu sein, den Interrupt immer zur selben Zeit auftreten zu lassen - obwohl die Interruptroutine immer einen konstanten Wert an Taktzyklen verbraucht, und deshalb der Flackereffekt gar nicht auftreten dürfte!
Tatsächlich liegt die Ursache allen Übels nicht beim Interrupt, sondern am Hauptprogramm: Tritt nämlich eine Interruptanforderung am Prozessor auf, so muß dieser zunächst einmal den aktuell bearbeiteten Befehl zu Ende führen, bevor er den Interruptvektor anspringen kann.
Angenommen, er wäre gerade dabei den Befehl "LDA #$00" auszuführen. Dieser Befehl benötigt 2 Taktzyklen. Einen zum Lesen und Dekodieren des Befehlsbytes, und einen zum Lesen des Operanden und Laden des Akkus. Hat der Prozessor nun gerade das Befehlsbyte gelesen und tritt in genau diesem Moment der Interrupt auf, so muß er zunächst noch den Operanden lesen, um anschließend in die IRQ-Routine verzweigen zu können. Selbige wird dadurch aber erst einen Taktzyklus später, als eigentlich erforderlich gewesen wäre, ausgeführt. Noch größer wird die Verzögerung, wenn gerade z. B. ein "STA $D021" (4 Taktzyklen!) oder gar ein "ROR $1000,X" (7 Taktzyklen!) ausgeführt wurde.
Das Programm "FLACKER" auf dieser MD veranschaulicht dieses Problem. Die Routine ist eine Kombination aus FLD und Borderroutine. Mit dem FLD-Teil drücken wir einen Teil des Bildschirms nach unten und stellen im Zwischenraum einen Rasterbalken dar. Der

Border-Teil schaltet einfach den oberen und unteren Bildschirmrand weg und dient mehr als Beispiel zur Kombination der beiden Rastereffekte. Der Rasterbalken ist nun auch in der X-Richtung verschiedenfarbig, so daß eine Zeile in der einen Hälfte eine Farbe und in der anderen Hälfte eine zweite Farbe enthält. Dieser "Rastersplit" ist übrigens nur durch die FLD-Routine möglich, da diese ja verhindert, daß eine Charakterzeile gelesen wird, die den Prozessor für 42 Taktzyklen (zwei Drittel der gesamten Zeile also) anhält. Ohne FLD könnte zum Beginn jeder Charakterzeile die Farbänderung gar nicht rechtzeitig (nämlich nach der Hälfte) stattfinden, da der Prozessor zum erforderlichen Zeitpunkt ja immer noch durch den VIC angehalten wäre. Außerdem hat die FLD-Routine den Vorteil, daß wir keinen Unterschied zwischen Charakter- und normalen Rasterzeilen machen müssen, weshalb wir uns die verschachtelten Schleifen sparen. Sie starten das Programm wie immer mit "SYS4096" und verlassen es durch einen Druck auf den Feuerknopf. Mit Joystickbewegungen nach oben und unten können Sie übrigens die Größe der FLD-Öffnung variieren.
Kommen wir nun zu unserem Problem zurück: sieht man sich das Beispielprogramm einmal an, so wird man ein erhebliches Flackern bemerken. Das liegt daran, daß das Programm zur Demonstration gerade in den zeitkritischen Momenten, besonders zeitintensive Befehle ausführt, weswegen der IRQ mit bis zu 7 Taktzyklen Verzögerung auftreten kann.
Da nun aber für die weiteren Beispiele dieses Kurses eine höhere Präzision erforderlich ist, müssen wir uns eine Methode angewöhnen, mit der wir einen Interrupt "glätten" können. Selbiges tut nämlich das dritte Beispiel dieses Kursteils, mit dem Namen "LOESUNG". Hier das dokumentierte Listing:

```
;*** Initialiserung ($1000)
        Init:       SEI             ;IRQ sperren
                    LDA #$7F        ;Timer IRQ
                    STA $DC0D       ;abschalten
                    BIT $DC0D       ;ICR löschen
                    LDA #$F8        ;Rasterzeile $f8 als
                    STA $D012       ;IRQ-Auslöser festlegen
                    LDA $d011       ;Bit 7 löschen
                    AND #$7F
                    STA $D011
                    LDA #$01        ;Raster als IRQ
                    STA $d01a       ;wählen
                    LDX #<Bord      ;Hard-IRQ-Vektoren
                    LDY #>Bord      ;auf eigene
                    STX $FFFE       ;Routine
                    STY $FFFF       ;umstellen
                    LDA #$33        ;Zeilenabstand für FLD
                    STA $02         ;initialisieren
                    LDA #$00        ;VIC-Byte löschen
                    STA $3FFF
                    LDA #$35        ;ROM ausblenden
                    STA $01
                    CLI             ;Interrupts erlauben
```

Hier haben wir den Initialisierungsteil vor uns. Wie üblich sperren wir zunächst die IRQs mittels SEI-Befehl, schalten alle von CIA-A möglichen Interruptquellen ab, und löschen mit Hilfe des BIT-Befehls eine evtl. noch gemeldete Interruptanfrage. Anschließend wird Rasterzeile $F8 als Interruptauslöser festgelegt (mit Löschen des Hi-Bits in $D011) und dem VIC mitgeteilt, daß er Raster-IRQs erzeugen soll. Nun erst wird der IRQ-Vektor (wohlgemerkt der bei $FFFE/$FFFF und nicht der bei $0314/$0315) mit der Adresse der Border-Routine gefüttert. Zum Schluß löschen wir dann noch die letzte VIC-Adresse,

deren Inhalt ja im abgeschalteten Border und dem FLD-Bereich angezeigt wird, und legen den FLD-Zähler fest (Speicherzelle $02), so daß er $33 (dez.51) Zeilen öffnet. Nun erst wird das ROM durch Schreiben von $35 in die Speicherzelle 1 ausgeblendet, damit der Prozessor bei einem IRQ auch unseren Vektor bei $FFFE/$FFFF anspringt, und die Interrupts werden wieder erlaubt. Die Init-Routine kehrt nun nicht wieder zur normalen Eingabe zurück, da wir damit ja in eine Routine des Basics zurückspringen würden, die nach abschalten des ROMs nicht mehr vorhanden ist. Stattdessen folgt nun eine Hauptschleife, mit der wir ständig den Joystickknopf abfragen. Hierbei erfüllen die ersten sieben Befehle eigentlich keinen Zweck. Es sind nur besonders zeitintensive Befehle, die wir zur Überprüfung, ob unser Glätten auch funktioniert, im Programm haben. Sie sind gefolgt von einer simplen Abfrage des Feuerknopf-Bits von Joyport 2:

```
;*** Hauptprogramm
        wfire   INC $03         ;Dummy-Befehle, die
                INC $2000       ;absichtlich besonders
                ROR $03         ;viel Rechenzeit
                ROR $2000       ;verbrauchen.
                BIT $03
                LDX #$00
                ROR $2000,X
                LDA $ DC00      ;Joyport laden
                AND #$10        ;Firebutton-Bit isol.
                BNE wfire       ;Nicht gedrückt -> weiter
```

Wird der Feuerknopf nun gedrückt, so müssen wir die ROMs wieder einschalten, dem VIC die IRQs verbieten und sie der CIA wieder erlauben, um das Programm verlassen zu können. Dies tut folgende Endroutine:

```
                SEI             ;IRQs sperren
                LDA #$37        ;ROMs einschalten
                STA $01
                LDA #$F0        ;VIC-IRQs sperren
                STA $D01A
                DEC $D019       ;ggf.VIC-IRQ-Anf.lösch.
                LDA #$1B        ;Normaler Darstellungs-
                STA $D011       ; modus (wg. Border)
                LDA #$81        ;CIA-A darf Timer-IRQs
                STA $DC0D       ; auslösen
                BIT $DC0D       ;ggf.CIA-IRQ-Anf.lösch.
                CLI             ;IRQs freigeben
                RTS             ;und ENDE
```

Kommen wir nun zur Borderroutine. Sie ist die erste IRQ-Routine, die nach der Initialisierung (bei Rasterzeile $F8) aufgerufen wird:

```
        Bord    PHA             ;Akku, X- u. Y-Register
                TXA             ;auf Stapel retten
                PHA
                TYA
                PHA
                LDA #$10        ;24-Zeilen-Darstellung an
                STA $D011       ;(=Bordereffekt)
                LDA #$3D        ;nächsten IRQ bei 2.
                STA $D012       ;Charakterzeile auslesen
                DEC $D019       ;VIC-ICR löschen
                LDX #<FLD1      ;IRQ-Vektoren auf
                LDY #>FLD1      ;erste
                STX $FFFE       ;FLD-Routine
                STY $FFFF       ; verbiegen
```

```
                JSR JoyCk       ;Joystickabfrage
                PLA             ;Akku, X- u. Y-Register
                TAY             ; wieder vom Stapel
                PA              ;holen
                TAX
                PLA
                RTI             ;IRQ beenden.
```

Die Routine macht eigentlich nichts weiter, als die 24-Zeilen-Darstellung zu aktivieren, die in Rasterzeile $F8 ja das Abschalten des Borders bewirkt, die Rasterzeile des nächsten IRQs festzulegen ($3D= Startposition der 2. Charakterzeile), den Interrupt-Vektor auf die Routine (FLD) für diese Zeile zu verbiegen, und den Joystick abzufragen (Unterroutine, die hier nicht aufgeführt ist).
Beachten Sie, daß wir hier die Prozessorregister mit Hilfe der Transfer und Stapelbefehle von Hand retten und wiederherstellen müssen. Gerade das Retten war nämlich eine Aufgabe, die uns das Betriebssystem freundlicherweise schon abgenommen hatte. Da es jetzt ja abgeschaltet ist, müssen wir uns natürlich selbst darum kümmern.

Kommen wir nun zur ersten FLD-Routine.
In ihr wird der Interrupt geglättet, was eine besonders trickreiche Angelegenheit ist. Sehen Sie sich hierzu einmal den Sourcecode an:

```
;*** FLD-Routine mit Glättung ($1100)
        FLD1    PHA             ;Akku, X- u. Y-Register
                TXA             ;retten
                PHA
                TYA
                PHA
                DEC $D019       ;neue IRQs erlauben
                INC $D012       ;nächte Raster.=Ausl.
                LDA #<FLD2      ;Low-Byte von FLD-IRQ2-
                STA $FFFE       ;Routine setzen
                CLI             ;IRQs erlauben
        WIRQ    NOP             ;13 NOPs
                NOP             ;(innerhalb dieser
                NOP             ;Schleife wird der
                NOP             ;Interrupt ausge-
                NOP             ;löst werden!!)
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                JMP QIRQ
        FLD2    PLA             ;Programmzähler und
                PLA             ;Statusreg. gleich
                PLA             ;wieder vom.Stack holen
                NOP             ;19 NOPs zum Verzögern
                NOP             ;bis zum tatsächlichen
                NOP             ;Charakterzeilen-
                NOP             ; anfang
                NOP
                NOP
                NOP
                NOP
```

```
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                LDA $D012       ;Den letzten Zyklus
                CMP $D012       ;korrigieren
                BNE cycle
cycle …                         ;eigentlicher IRQ
```

Hier werden Ihnen einige Dinge etwas merkwürdig vorkommen (vor allem die vielen NOPs). Beginnen wir von Anfang an:
In der Borderroutine hatten wir die Rasterzeile festgelegt, in der die FLD1-Routine angesprungen werden soll. Dies war Zeile 61($3D), die genau zwei Rasterzeilen vor der eigentlichen IRQ- Rasterzeile liegt. In diesen zwei Zeilen, die wir den IRQ früher ausgelöst haben, werden wir ihn jetzt glätten. Wie in jedem Interrupt retten wir zunächst die Prozessorregister. Daran anschließend wird das Low-Byte der Routine "FLD2" in das Low-Byte des IRQ-Vektors geschrieben, und die nächste Rasterzeile (durch den INC-Befehl) als nächster Interruptauslöser festgelegt. Beachten Sie hierbei, daß die diese Routine dasselbe High-Byte in der Adresse haben muß, wie die erste FLD-Routine. Das kann man dadurch erzielen, daß "FLD1" an einer Adresse mit 0-Lowbyte ablegt wird und sofort danach die Routine "FLD2" folgt (im Beispiel ist FLD1 an Adresse $1100).
Um einen neuen Interrupt zu ermöglichen, müssen noch das ICR des VIC und das Interrupt-Flag des Prozessors gelöscht werden (beachten Sie, daß letzteres vom Prozessor automatisch bei Auftreten des IRQs gesetzt wurde).
Es folgt nun der eigentliche "Glättungsteil". Hierzu lassen wir den Prozessor ständig durch eine Endlos-Schleife mit NOP-Befehlen laufen. Dadurch wird sichergestellt, daß der Raster- IRQ der nächsten Rasterzeile in jedem Fall während der Ausführung eines NOP-Befehls auftritt. Da dieser Befehl nur 2 Taktzyklen verbraucht, kann die Verzögerung des Interupts nur 0 oder 1 Taktzyklen lang sein. Diesen einen Zyklus zu korrigieren ist nun die Aufgabe des zweiten FLD-IRQs. Nachdem er angesprungen wurde holen wir gleich wieder die, vom Prozessor automatisch gerettete, Programmzähleradresse und das Statusregister vom Stapel, da sie nur zum ersten FLD-Interrupt gehören, in den nicht mehr zurückgekehrt werden soll.
Danach folgen 19 NOP-Befehle, die nur der Verzögerung dienen, um das Ende der Rasterzeile zu erreichen. Die letzten drei Befehle sind die Trickreichsten!
Sie korrigieren den einen möglichen Verzögerungs-Zyklus. Zum besseren Verständnis sind sie hier nochmal aufgelistet:

```
                LDA $D012       ;Den letzten Zyklus
                CMP $D012       ;korrigieren
                BNE cycle
cycle ...                       ;eigentlicher IRQ
```

Obwohl diese Folge recht unsinnig erscheint, hat sie es ganz schön in sich:
Wir laden hier zunächst den Akku mit dem Inhalt von Register $D012, das die Nummer der aktuell bearbeiteten Rasterzeile beinhaltet, und vergleichen ihn sofort wieder mit diesem Register. Danach wird mit Hilfe des BNE-Befehls auf die Folgeadresse verzweigt, was noch unsinniger erscheint.

Der LDA-Befehl befindet sich nun durch die NOP-Verzögerung genau an der Kippe zur nächsten Rasterzeile, nämlich einen Taktzyklus bevor diese Zeile beginnt.
Sollte nun der FLD2-IRQ ohne den einen Taktzyklus Zeitverzögerung ausgeführt worden sein, so enthält der Akku z.B. den Wert 100. Der CMP-Befehl ist dann ein Vergleich mit dem Wert 101, da der Rasterstahl nach dem LDA-Befehl schon in die nächste Zeile gesprungen ist. Dadurch sind die beiden Werte ungleich, womit das Zero-Flag gelöscht ist, und der Branch tatsächlich ausgeführt wird.
Beachten Sie nun, daß ein Branch-Befehl bei zutreffender Bedingung durch den Sprung einen Taktzyklus mehr Zeit verbraucht, als bei nicht zutreffender Bedingung (3 Zyklen!). War der FLD2-IRQ allerdings mit dem einem Taktzyklus Verzögerung aufgetreten, so wird der LDA-Befehl genau dann ausgeführt, wenn Register $D012 schon die Nummer der nächsten Rasterzeile enthält, womit der Akku den Wert 101 beinhaltet. Durch den Vergleich mit dem Register, das dann immer noch den Wert 101 enthält, wird das Zero-Flag gesetzt, da die beiden Werte identisch sind. Dadurch trifft die Bedingung des BNE-Befehls nicht zu, und er verbraucht nur 2 Taktzyklen! Dies gewährleistet, daß in beiden Fällen immer die gleiche Zyklenzahl verbraucht wird! War der FLD2-IRQ ohne Verzögerung, so verbracht die Routine einen Zyklus mehr, als wenn er mit einem Zyklus Verspätung auftrat!
Hiermit hätten wir den IRQ also geglättet und können die eigentliche FLD und Farbsetz-Routine ausführen. Beachten Sie für Folgebeispiele, daß wir in Zukunft auf diese Weise die IRQs immer glätten werden müssen, um saubere Ergebnisse zu erzielen. Hierzu wird immer wieder diese Routine verwandt, wobei das eigentliche IRQ-Programm dann nach dem Branch-Befehl eingesetzt wird. Gleichmäßiger kann man Raster-IRQ nun wirklich nicht mehr ausführen!
Nach dem Glätten folgt die eigentliche Interruptroutine, und zwar direkt nach dem Label "Cycle". Sie setzt Rasterzeile $F8 als Interruptauslöser fest und verbiegt den IRQ-Vektor wieder auf die Borderroutine, womit der Kreislauf von Neuem beginnt. Gleichzeitig setzt Sie die Darstellung auf 25 Zeilen zurück, damit der Bordereffekt auch funktioniert. Anschließend wird der FLD-Effekt durchgeführt, indem der Zeilenanfang vor dem Rasterstrahl hergeschoben wird. Währenddessen werden die Vorderund Hintergrundfarbe nach zwei Farbtabellen bei $1200 und $1300 verändert. Der Rastersplit wird durch ausreichende Verzögerung bis zur Mitte einer Rasterzeile erzeugt. Zum Schluß des IRQs wird noch bis zum Ende der Rasterzeile verzögert, und die Standardfarben zurückgesetzt, bevor die ursprünglichen Inhalte der Prozessorregister, wie sie vor dem Auftreten des FLD1- IRQs vorhanden waren, zurückgeholt werden und der IRQ beendet wird:

```
Cycle       DEC $D019       ;VIC-ICR löschen
            LDA #$18        ;25- Zeilen-Darstellung ein-
            STA $D011       ;schlaten (Bordereffekt)
            LDA #$F8        ;Rasterz. $F8 ist näch-
            STA $D012       ;ster IRQ-Auslöser
            LDX #<Bord      ;IRQ-Vektor
            LDY #>Bord      ;auf
            STX $FFFE       ;Border-Routine
            STY $FFFF       ;verbiegen
            NOP             ;Verzögern
            LDA $02         ;FLD-Zähler laden
            BEQ FDEnd       ;Wenn 0, kein FLD!
            LDX #$00        ;Zählregister für Farben initialiseren
            CLC
FDLop       LDA $D012       ;FLD-Sequenz (Zeilen-
            ADC #$02        ;anfang vor Raster-
            AND #$07        ;strahl herschieben
            ORA #$18
```

```
                STA $D011
                LDA $1200,X     ;Farbe links holen
                STA $D020       ;und setzen
                STA $D021
                NOP             ;Verzögern bis Mitte
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                LDA $1300,X     ;Farbe rechts holen
                STA $D020       ;und setzen
                STA $D021
                BIT $EA         ;Verzögern
                INX             ;Farb-Zähler+1
                CPX $02         ;Mit FLD-Zähler vgl.
                BCC FDLop       ;ungl., also weiter
FDEnd           NOP             ;Verz. bis Zeilenende
                NOP
                NOP
                NOP
                NOP
                LDA #$0E        ;Vorder-/Hintergrund-
                STA $D020       ;farben auf
                LDA #$06        ;hellblau u. dunkel-
                STA $D021       ;blau setzen
                PLA             ;Akku, X- und Y-Register
                TAY             ;zurückholen
                PLA
                TAX
                PLA
                RTI             ;IRQ beenden
```

Das war es dann wieder für diesen Monat.
Im nächsten Kursteil werden wir eine weitere Anwendung besprechen, die eine IRQ-Glättung benötigt: die Sideborderroutinen zum Abschalten des linken und rechten Bildschirmrands, nämlich.

(ub)

Teil 7 – Magic Disk 05/94

Nachdem wir uns im letzten Kursteil ausgiebig um das IRQ-Timing gekümmert hatten, wollen wir in dieser Ausgabe nun eine Anwendung besprechen, bei der vor allem das Glätten von Interrupts eine große Bedeutung einnimmt: es geht um die Sideborderroutinen.

1. Das Prinzip

Im dritten Kursteil hatten wir ja schon gelernt, wie man den oberen und unteren Bildschirmrand abschaltet. Wir hatten dem VIC hierzu zunächst mitgeteilt, daß er einen 25-Zeilen hohen Bildschirm darstellen soll. Zwei Rasterzeilen bevor er jedoch den Rand desselben erreichte schalteten wir ihn auf 24- Zeilen-Darstellung um, weswegen er glaubte, schon zwei Rasterzeilen vorher mit dem Zeichnen des Randes begonnen zu haben.
Da dies aber nicht der Fall war, und der VIC seine Arbeit normal fortführte,"vergaß" er

sozusagen, den unteren und den oberen Bildschirmrand zu zeichnen, was uns ermöglichte, in diesen Bereichen Sprites darzustellen. Derselbe Trick funktioniert nun auch mit dem linken und rechten Bildschirmrand - zumindest vom Prinzip her. Bit 3 in VIC-Register $D016 steuert die Breite des sichtbaren Bildschirms. Ist dieses Bit gesetzt, so zeichnet der VIC 320 sichtbare Pixel in der Vertikalen, was einer Darstellung von 40 Zeichen pro Zeile entspricht.
Löschen wir dieses Bit, so stellt er nur 302 Pixel, bzw.38 Zeichen pro Zeile dar. Dieses Bit wird vor allem zum vertikalen Scrollen verwendet, da man bei einem 38-Spalten-Bildschirm neu hereinlaufende Bildschirmzeichen setzen kann, ohne daß sie vom Betrachter gesehen werden.
Durch das rechtzeitige Setzen und Löschen dieses Bits kann man nun auch den linken und rechten Bildschirmrand abschalten. Hierbei liegt die Betonung besonders auf dem Wort "rechtzeitig". Da der Rasterstrahl in der vertikalen nämlich derart schnell ist, daß der Prozessor kaum nachkommt, müssen wir den Zeitpunkt der Umschaltung sehr genau abpassen, damit unser Effekt funktioniert.
Bei der 38-Zeichen-Darstellung beginnt der linke Bildschirmrand nur 8 Pixel später und endet nur 8 Pixel früher als sonst. Da aber genau dann, wenn er Endet, umgeschaltet werden muß, und der Rasterstrahl zum Zeichnen von 8 Pixeln gerade mal 1,2 Taktzyklen benötigt, haben wir eben nur genau diese Zeit zur Verfügung, um Bit 3 in Register $D016 zu löschen. Da ein "STA$D016" aber drei Taktzyklen verbraucht, muß der Prozessor diesen Befehl also schon zwei Zyklen vor dem Bildschirmrand erreichen und beginnen abzuarbeiten. Der eigentliche Schreibzugriff findet dann im dritten Taktzyklus, genau zwischen 38- und 40-Zeichen-Rand statt. Zur Verdeutlichung sollten Sie sich einmal das Programmbeispiel "SIDEBORDER.0" anschauen. Wie alle unsere Beispiele ist es absolut (also mit ",8,1") zu laden und mit SYS4096 zu starten. Sie verlassen die Routine mit einem Druck auf den Feuerknopf eines Joysticks in Port2. Mit Joystickbewegungen nach oben und unten können Sie die Anzahl der zu öffnenden Zeilen variieren.
Wir möchten Ihnen nun die Kernroutine des Beispiels zeigen. Die Erklärung der Interruptinitialisierung werden wir uns an dieser Stelle sparen, da wir sie ja schon oft genug besprochen haben. Im Beispiel selbst haben wir alle Interruptquellen, außer den Interrupts vom Rasterstrahl gesperrt. Des weiteren wurde das Betriebssystems-ROM abgeschaltet, und die Adresse auf unsere Routine gleich in den Vektor bei $FFFF/$FFFE geschrieben, so daß der Prozessor ohne Zeitverzögerung zu unserer Routine springt. Selbige "glättet" den Interrupt zunächst nach der im letzten Kursteil beschriebenen Art und Weise. Direkt nach der Glättung folgt nun diese Routine:

```
                ...
                LDX #00         ;Zeilenzähler löschen
                CLC
LOOP            NOP             ;7 NOPs Verzögerung
                NOP             ;bis Zeilenende
                NOP
                NOP
                NOP
                NOP
                NOP
                LDA #00         ;EINEN Taktzykl. vor
                STA $D016       ;Beg. d.rechten Randes
                LDA #08         ;Bit 3 löschen und
                STA $D016       ;gleich wieder setzen
                NOP             ;13 weitere NOPs zum
                NOP             ;Verzögern von 26
                NOP             ;Taktzyklen
                NOP
```

```
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                BIT $EA         ;3 Zyklen verzögern
                INX             ;Linienzähler+1
                CPX $02         ;Mit Anz.zu öffnender
                BCC LOOP        ;Zeilen vgl. u. weiter
                …
```

Diese Abfolge von Befehlen beinhaltet nun haargenau das benötigte Timing, damit der Sideborder-Effekt auch sichtbar wird. Sie sehen hier zunächst 7 NOPs, mit denen wir vom Ende der Rasterglättung bis zum Beginn des sichtbaren rechten Bildschirmrandes verzögern. Hieraufhin wird Bit 3 von Register $D016 gelöscht, um auf 38-Zeilen-Darstellung umzuschalten, und gleich darauf wieder gesetzt, damit das Umschalten in der nächsten Rasterzeile noch genauso effektiv ist. Anschließend wird mit den 13 NOPs und dem BIT-Befehl um insgesamt 29 Taktzyklen verzögert, und geprüft, ob das X-Register, das als Zähler der schon geöffneten Rasterzeilen fungiert, den Wert der zu öffnenden Zeilen enthält, der in Speicherzelle $02 abgelegt ist.
Wenn nicht, so wird wieder zum Anfang der Schleife verzweigt und somit eine weitere Rasterzeile geöffnet.

2. Eine Verfeinerung

Wenn Sie das Beispiel einmal gestartet und angesehen haben, so werden Sie bemerken, daß wir lediglich im oberen, sonst mit dem Rahmen verdeckten Bereich des Bildschirms den seitlichen Rand geöffnet haben. Das funktioniert natürlich nur, wenn wir den VIC zuvor mit dem normalen Bordereffekt überlistet haben, so daß er den oberen und unteren Bildschirmrand ebenfalls wegfallen lässt.
Des weiteren kann der Rand dort nur bis zur letzten Zeile, des (unsichtbaren) oberen Bildschirmrandes geöffnet werden, was einen besonderen Grund hat: Wie wir mittlerweile ja wissen, liest der VIC zum Beginn einer jeden Charakterzeile (also jede achte Rasterzeile), die 40 in dieser Zeile darzustellenden Zeichen ein, und blockiert während dieser Zeit den Prozessor für eine Zeitspanne von 42 Taktzyklen. Dadurch gerät natürlich unser ganzes Timing durcheinander, weshalb der Effekt an diesen Stellen nicht mehr zu sehen ist (obwohl immer noch dieselbe Schleife läuft!). Um nun innerhalb des normalen Bildschirms, in dem Zeichen dargestellt werden können, den Rand zu öffnen, verkompliziert sich das Timing natürlich erheblich, da wir jede achte Rasterzeile 42 Taktzyklen weniger zur Verfügung haben. Die einfachste Methode, dieses Problem zu umgehen, ist die Kombination der Sideborderroutine mit einer FLD-Routine. Dadurch schieben wir ja den Beginn der nächsten Charakterzeile vor dem VIC her, so daß er keine Daten liest und uns somit nicht bei der Arbeit stört. Diesen Weg geht unser zweites Programmbeispiel mit dem Namen "SIDEBOR-DER.1". Hier das Listing der entscheidenden Schleife. Auch hier wurde der Interrupt natürlich zuvor geglättet:

```
                ...
LOOP            LDA $D012       ;FLD-Routine
                ADC #$02
                AND #$07
                ORA #$18
```

```
STA $D011
LDA #$00          ;wieder 1 Zykl. vor Beg.
STA $D016         ;des rechten Randes Bit 3
LDA #$08          ;von $D016 löschen und
STA $D016         ;gleich wieder setzen
NOP               ;13 weitere NOPs zum
NOP               ;Verzögern von 26
NOP               ;Taktzyklen
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
BIT $EA           ;3 Zyklen verzögern
INX               ;Linienzähler+1
CPX $02           ;Mit Anz. zu öffnender
BCC LOOP          ;Zeilen vgl. u. weiter
…
```

Im Prinzip hat sich hier nicht viel geändert. Nur daß die 7 NOPs am Anfang der Routine den fünf Befehlen des FLD-Effektes gewichen sind. Diese brauchen ebenso 14 Taktzyklen, so daß unsere Routine vom Timing her immer noch genauso funktioniert, wobei wir den Bildschirmrand jedoch innerhalb des normalen Bildbereichs abgeschaltet haben!

3. Sprites im Sideborder

Wie bei den Bildschirmrändern oben und unten kann man natürlich auch in den seitlichen, abgeschalteten Rändern Sprites darstellen. Dies gestaltet sich jedoch ebenfalls schwieriger als sonst.
Nämlich so wie der VIC den Prozessor in jeder Charakterzeile anhält, um ungestört die Charakterdaten zu lesen, so hält er ihn auch an, wenn er Spritedaten zu lesen hat. Hierbei prüft der VIC zunächst einmal, ob ein Sprite an der aktuellen Rasterposition überhaupt sichtbar ist. Wenn ja, so liest er die drei Datenbytes des Sprites, die in dieser Rasterzeile anzuzeigen sind. Hierzu benötigt er ungefähr 2,4 Taktzyklen, die uns von der Prozessorzeit natürlich wieder abgehen! Somit muß die Verzögerung innerhalb unserer Schleife verkürzt werden. Da wir aber keine Bruchteile von Taktzyklen verzögern können, wird die ganze Sache umso haariger!!! In der Regel hilft hier nur die alte "Trialand- Error"-Methode, um das perfekte Timing genau auszuloten. Zum Einschalten von sieben Sprites müssen wir nach dem Löschen und Setzen des Anzeigebits von Register $D016 nur noch 12 Takte, anstelle von ursprünglich 29, warten, bis der Rasterstrahl die nächste Zeile erreicht. Es gehen uns also 17 Zyklen verloren! Programmbeispiel "SIDEBORDER.2" benutzt anstelle der 13 NOPs und dem BIT-Befehl nur noch 6 NOPs. Sehen Sie sich dieses Beispiel ruhig einmal mit einem Disassembler oder Speichermonitor an. Wenn Sie durch Herunterdrücken des Joysticks den geöffneten Bereich vergrößern, so werden Sie merken, daß der Sideborder-Effekt nur in den Rasterzeilen funktioniert, in denen die Sprites sichtbar sind. Das gilt auch dann, wenn Sie die Sprites in Y-Richtung expandieren (vor dem Aufruf des Beispiels POKE53271,255 eingeben). Hier funktioniert der Effekt dann doppelt so viele Zeilen lang, da der VIC bei Expansion jede Zeile eines Sprites einfach doppelt liest (auf zwei Rasterzeilen verteilt). Möchten Sie den Rand weiter nach unten öffnen, so ist nur noch ein komisches Linien-Wirr-Warr zu sehen, da in diesen Zeilen keine

Sprites angezeigt werden, und somit wieder eine Verzögerung von 29 Taktzyklen von Nöten wäre! Wohlgemerkt funktioniert dieser Effekt nur, wenn auch wirklich in jeder Rasterzeile sieben Sprites darzustellen sind. Schalten Sie weniger oder mehr Sprites ein, so ist der Sideborderеffekt zunichte gemacht. Das gleiche passiert, wenn Sie nicht alle sieben Sprites Y-expandieren.
Hier Endet der Effekt nach 21 Zeilen, nämlich dann wenn ein nicht expandiertes Sprite zu Ende gezeichnet ist, und Sie sehen im Border nur die Hälfte der expandierten Sprites!

4. Hinweise zum optimalen Timing
Möchten Sie nun selbst die Verzögerung ausloten, die zur Darstellung einer bestimmten Anzahl Sprites benötigt wird, so beachten Sie folgende Regeln:

1. Geben Sie allen Sprites, die in Bereichen angezeigt werden, in denen der Sideborder abgeschaltet ist, ein und dieselbe Y-Koordinate, so daß sie horizontal in einer Linie liegen. Dadurch werden Ungleichheiten innerhalb der Rasterzeilen vermieden, so daß immer gleich viele Spritedaten gelesen werden. Gleiches gilt für die Y-Expandierung. Es sollten entweder alle oder keines der, in einem solchen Bereich sichtbaren, Sprites expandiert sein (es sei denn Sie möchten nur die Hälfte eines expandierten Sprites sehen).

2. Achten Sie darauf, daß andere Sprites nicht in den Bereich ohne Sideborder hineinragen, da auch sie das Timing durcheinanderbringen.

3. Beim Ausloten der Verzögerung sind NOPund BIT-Befehle am Besten zum schrittweisen Timing-Test geeignet.
   Möchten Sie eine ungerade Anzahl Zyklen verzögern, so benutzen Sie einen Zeropage-BIT- Befehl (so wie der oben benutzte "BIT $ EA"), da er drei Zyklen verbraucht. Bei einer geraden Anzahl Zyklen reicht eine entsprechende Anzahl NOPs, die jeweils nur 2 Zyklen benötigen. So brauchen Sie zur Verzögerung von z.B. 7 Zyklen 2 NOPs und einen BIT-Befehl. Bei 8 Zyklen sind 4 NOPs ohne BIT-Befehl ausreichend, usw.

4. Schätzen Sie in etwa ab, wie viele Taktzyklen Sie bei einer bestimmten Anzahl Sprites etwa benötigen, um zu verzögern (Formel: 29-AnzahlSprites*2,4) und testen Sie ob dieser Wert funktioniert. Wenn nicht ändern Sie in 1-Zyklus-Schritten nach oben oder unten.

5. Natürlich wird der Prozessor immer nur eine gerade Anzahl Zyklen angehalten. Der Richtwert von 2,4 Zyklen pro Sprite ist lediglich zur Orientierung gedacht. Da der Prozessor immer nur zu einem Taktsignal eine Operation beginnen kann, sollte die Verzögerung also immer zu finden sein.

6. Zum Testen, wann Register $D016 geändert wird, sollten Sie die Zugriffe auf dieses Register zu Testzwecken mit Zugriffen auf Register $D021 ersetzen.
   Sie erzielen dadurch eine Farbänderung des Hintergrunds, und zwar genau zu dem Zeitpunkt, zu dem normalerweise auch Register $D016 verändert wird.
   Sehen Sie einen schwarzen Strich am Bildschirmrand, der länger als 8 Pixel ist, so ist der Zugriff zu früh. Sehen Sie gar keinen Strich, so war er zu spät. Wenn Sie die richtige Einstellung gefunden haben können Sie $D021 wieder mit $D016 ersetzen, und der Sideborder-Effekt sollte funktionieren.

5. Noch trickreichere Programmierung

Zum Abschluß möchte ich Ihnen noch ein viertes Programmbeispiel erläutern, nämlich eine Sideborderroutine, die acht Sprites im Border darstellt. Das verwunderliche an ihr ist die Funktionsweise.
Denn obwohl die Sideborderschleife genauso viele Taktzyklen verbraucht wie die Version für sieben Sprites, kann dennoch ein Sprite mehr dargestellt werden. Ermöglicht wird dies durch eine trickreichere Variante des Löschen und Setzen des Bildschirmbreite-Bits aus Register $D016. Wir initialisieren dieses Register vor der eigentlichen Sideborderroutine nämlich mit dem Wert $C8, was dem Standardwert dieses Registers entspricht. In der eigentlichen Routine wird dann Bit 3 gelöscht, indem wir das ganze Register mittels "DEC $D016" um eins herunterzählen (auf $C7 - bin. %11000111 - Bit 3 gelöscht). Ein direkt folgendes "INC $D016" setzt das Bit dann wieder. Hier das Listing der Kernroutine des Beispiels "SIDEBORDER.3":

```
                ...
LOOP            LDA $D012       ;FLD-Routine
                ADC #$02
                AND #$07
                ORA #$18
                STA $D011
                DEC $D016       ;Bit 3 löschen und
                INC $D016       ;gleich wieder setzen
                NOP             ;Wie bei "SIDEBORDER.2"
                NOP             ;stehen hier trotzdem
                NOP             ;nur 6 NOPs!!!
                NOP
                NOP
                NOP
                INX             ;Linienzähler+1
                CPX $02         ;Mit Anz. zu öffnender
                BCC LOOP        ;Zeilen vergleichen und weiter
                …
```

Der Grund, warum die Routine dennoch funktioniert ist, daß der Prozessor zur Abarbeitung des DEC-Befehls sechs Taktzyklen verbraucht, wobei der Schreibzugriff mit dem dekrementierten Wert durch den internen Aufbau des Befehls schon früher eintritt. Alles in Allem ist das Funktionieren dieser Methode umso wunderlicher, da die beiden Befehle DEC und INC, ebenso wie die beiden LDAs und STAs vorher,12 Taktyklen benötigen. Wir tauchen hier also schon in die tiefere Bereiche der Assemblerprogrammierung ein, da es an der Funktionsweise des DEC-Befehls liegt, warum das Bit rechtzeitig gelöscht wird.
Beachten Sie daher bitte auch diese Variation der Sideborder-Routine für eigene Zwecke und benutzen Sie sie, wenn normales Ausloten nicht ausreicht.

6. ein letztes Beispiel

Als zusätzlichen Leckerbissen haben wir Ihnen noch ein weiteres Beispielprogramm auf dieser MD untergebracht:"SIDEBOR-DER.4" basiert auf dem selben Prinzip wie "SIDEBORDER.3", nur daß die Sprites zusätzlich X-expandiert wurden und somit eine lückenlose Laufschrift durch den Bildschirmrand dargestellt werden kann.
Die Zeichendaten werden dabei direkt aus dem Zeichensatz-ROM geholt, und mit Hilfe des ROL-Befehls bitweise durch die Sprites rotiert. Wie so etwas funktioniert wissen Sie bestimmt, weshalb ich es hier nicht noch einmal extra erläutere, zumal es ja eigentlich nicht zu unserem Thema gehört.
Übrigens: vielleicht ist Ihnen schon aufgefallen, daß beim Öffnen des Sideborders der

rechte Bildschirmrand immer eine Rasterzeile höher geöffnet ist, als der Linke. Und das dieser umgekehrt eine Rasterzeile länger offen ist, als der Rechte. Das liegt daran, daß das Abschalten des Randes zwar am rechten Ende einer Rasterzeile geschieht, sich jedoch auf das linke Ende der folgenden Rasterzeile auswirkt!
In der nächsten Folge dieses Kurses werden wir Ihnen zeigen, wie man den Border OHNE wegdrücken des Bildschirms durch FLD abschaltet (besonders haarige Timingprobleme) . Des weiteren bleiben wir dann beim Thema "Sprites", und wollen uns anschauen, wie man den VIC derart austrickst, daß er mehr als acht dieser kleinen Grafikstückchen auf den Bildschirm zaubert. Bis dahin sollten Sie sich die in der heutigen Folge besprochenen Beispiele noch einmal genauer anschauen, und ein wenig damit herumexperimentieren. Sie ahnen nicht wie vielseitig man mit dem abgeschalteten Border arbeiten kann...

(ub/ih)

Teil 8 – Magic Disk 06/94

Herzlich willkommen zum achten Teil unseres Raster-IRQ-Kurses. Heute wollen wir uns mit einem besonders interessanten Rastertrick beschäftigen: der FLI-Routine, die es uns ermöglicht, Grafiken mit mehr als 2 Farben im Hires-Modus, bzw. mehr als 4 Farben im Multicolor-Modus, jeweils pro 8 x8- Pixel-Block, darzustellen. Durch die dabei entstehende Farbvielfalt können sehr eindrucksvolle Bilder angezeigt werden, die bei geschickten Grafikern schon an die Qualität von Amiga-Bildern reichen können!

1. Grundlagen

Wollen wir uns zunächst einmal anschauen, wie der VIC vorgeht, um eine Bitmap-Grafik auf den Bildschirm zu zaubern:
Zunächst einmal muß der Grafikmodus eingeschaltet werden, wobei die Lage der Bitmap im Speicher mitangegeben wird.
Nun zeigt der VIC also die Bits des angegebenen Speicherbereichs als einzelne Pixel auf dem Bildschirm an. Um diesen Pixeln nun auch noch Farbe zu verleihen, muß in zwei Fälle unterschieden werden:

1.1. Der Hires-Modus

Hier bestimmt ein Farbcode im Video-RAM, das im Textmodus zur Darstellung der Zeichen benutzt wird und sich normalerweise bei $0400 (dez.1024) befindet, die Farbe der Pixel innerhalb eines 8x8- Pixel-Blocks. So legt das erste Byte des Video-RAMs (als Zeichen ganz links oben), die Farbe für die 8x8, im Grafikmodus quasi "über" ihm liegenden, Pixel fest. Das zweite Byte ist für den nächsten 8x8- Pixel-Block zuständig, und so weiter. Da der VIC insgesamt nur 16 Farben kennt, sind dabei jeweils nur die unteren vier Bits eines Video-RAM- Bytes von Bedeutung. Die oberen vier sind unbenutzt und werden ignoriert.

1.2. Der Multicolor-Modus

In diesem Modus werden zwei nebeneinander liegende Pixel jeweils zu einem Farbcode zusammengefasst. Sind beide 0, so erscheint an ihrer Stelle die Hintergrundfarbe, sind beide auf 1, so wird für beide die Farbe aus einem äquivalenten Byte des Color-RAMs geholt (bei Adresse $D800 - dez.55296), das normalerweise zur Farbgebung der einzelnen Zeichen im Textmodus verwendet wird. Bei der Bitkombination "10" wird wieder das Lownibble (die unteren vier Bits) des Video-RAMs zur Farbgebung ausgewertet.
Bei der Bitkombination "10" sind die, im Hires-Modus unbenutzten, oberen vier Bits des Video-RAMs für die Pixelfarbe zuständig. Sie sehen also, daß das Video-RAM im Multicolormodus gleich zwei Farbwerte für Pixelkombinationen festlegt.

2. Das FLI-Prinzip

Soviel zur Darstellung einer Grafik.
Sicherlich ist Ihnen bei den obigen Ausführungen das häufige Auftreten des Begriffs "Video-RAM", bzw."Video-Map" aufgefallen. Und genau dieser Begriff ist der Schlüssel zu unserer FLI-Routine. Wie Sie vielleicht wissen, kann man die Lage der Video-Map, innerhalb des Adressierungsbereichs des VICs (im Normalfall von $0000-$3 FFF) in 1 KB-Schritten verschieben. Hierfür ist Register 24 des VICs (Adresse $D018 – dez. 53272) zuständig. Seine oberen 4 Bits geben die Lage der Video-Map an, also des RAM-Bereichs, der für die Darstellung der Textzeichen, bzw. im Grafikmodus der Farbzeichen, zuständig ist. Die unteren Bits von 0-3 bestimmen die Lage des Zeichengenerators, also des Speicherbereichs, in dem die Daten für den Zeichensatz ausgelesen werden. Dies soll uns hier jedoch nicht interessieren und sei nur nebenbei angemerkt.
Konzentrieren wir uns auf die Bits 4-7:
Sie bestimmen die Lage des Video-RAMs innerhalb des 16 KB-Bereichs des VICs.
Die im Folgenden angegebenen Adressen verstehen sich also als Offsets, die auf die Startadresse des 16 KB-Bereichs aufaddiert werden müssen:

Wert	Bits	Bereich (HEX)	Bereich (DEZ)
0	0000	$0000 - $03FF	0 - 1023
1	0001	$0400 - $07FF	1024 - 2047
2	0010	$0800 - $0BFF	2048 - 3071
3	0011	$0CFF - $0FFF	3072 - 4095
4	0100	$1000 - $13FF	4096 - 5119
5	0101	$1400 - $17FF	5120 - 6143
6	0110	$1800 - $1BFF	6144 - 7167
7	0111	$1CFF - $1FFF	7168 - 8191
8	1000	$2000 - $23FF	8192 - 9215
9	1001	$2400 - $27FF	9216 -10239
10	1010	$2800 - $2BFF	10240 -11263
11	1011	$2CFF - $2FFF	11264 -12287
12	1100	$3000 - $33FF	12288 -13311
13	1101	$3400 - $37FF	13312 -14335
14	1110	$3800 - $3BFF	14336 -15359
15	1111	$3CFF - $3FFF	15360 -16383

Obwohl die Video-Map nur 1000 Bytes lang ist, habe ich hier dennoch 1024-Byte-Bereiche angegeben. Das hat nämlich auch eine Bedeutung für den nächsten Kursteil.
Des weiteren wollen wir noch schnell klären, wie man den 16 KB-Bereich des VICs verschiebt, da die FLI-Routine eine Menge Video-Speicher benötigt (nämlich die vollen 16 KB), sollten wir den VIC in einem Bereich arbeiten lassen, in dem wir nicht auf Zeropage-Adressen und Sprungvektoren (die im ersten VIC-Bereich - von $0000-$3FFF - eine Menge Platz wegnehmen) Rücksicht nehmen zu müssen.
Der Adressbereich des VIC wird nun mit den untersten zwei Bits der Adresse $DD00 (dez.56576) angegeben (richtig: dies ist ein CIA-B Register, das dem VIC seinen Adressbereich vorschreibt). Hier eine Auflistung der möglichen Bitkombinationen und der Adressbereiche die sie aktivieren:

Wert	Bits	Bereich (HEX)	Bereich (DEZ)
3	11	$0000-$3FFF	0-16383
2	10	$4000-$7FFF	16384-32767
1	01	$8000-$BFFF	32768-49151
0	00	$C000-$FFFF	49152-65535

In unserem Programmbeispielen werden wir den VIC-Bereich nach $4000 verschieben, womit der Wert 2 in die Adresse $DD00 geschrieben werden muß. Die tatsächliche Adresse der Video-Map ist dann immer $4000 plus der oben angegebenen Offset-Adresse. Nachdem wir nun also die Grundlagen geklärt hätten, ist die Erklärung des Prinzips der FLI-Routine höchst simpel:
Zum Einen wissen wir, daß die Video-Map zur Farbgebung der Grafik verwendet wird. Zum Anderen haben wir gesehen, wie die Startadresse, aus der der VIC sich die Video-Map-Daten holt, verschoben werden kann. Was liegt nun also näher als zwei und zwei zusammenzuzählen, und eine Raster-IRQ- Routine zu schreiben, die in JEDER Rasterzeile eine andere Video-Map aktiviert? Die Folge dieser Operation wäre dann nämlich die Möglichkeit, einem 8x8- Pixel Block der Grafik in jeder Zeile eine neue Farbe zuzuteilen (im Multicolormodus sogar 2), so daß wir die 2-, bzw.4- Farbeinschränkung auf einen Bereich von 1x8 Pixeln reduzieren!

3. Die Umsetzung
Vom Prinzip her klingt das natürlich sehr einfach, jedoch stellt sich uns noch ein kleines Problem in den Weg:
Wie wir bei der Beschreibung des FLD-Effektes schon gelernt hatten, liest der VIC ja nur in jeder Charakterzeile (also jede achte Rasterzeile), die 40 Bytes, die er in den folgenden acht Rasterzeilen darzustellen hat. Somit würde ein einfaches Umschalten auf eine neue Video- Map nichts bewirken, da der VIC ja immer noch mit den 40 Zeichen, die er zu Beginn der Charakterzeile gelesen hat, die folgenden Zeilen darstellen würde - und das selbst bei umgeschalteter Video-Map. Also müssen wir ihn auch hier mit einem kleinen Raster-Trick "veräppeln".
Man kann den VIC nämlich auch dazu zwingen, eine Charakterzeile NOCHMALS zu lesen. Der anzuwendende Trick ist uns dabei gar nicht mal so unbekannt, denn er funktioniert ähnlich wie der FLD-Effekt. Bei diesem schoben wir den Anfang der nächsten Charakterzeile durch Hochsetzen der vertikalen Bildschirmverschiebung vor dem Rasterstrahl her, so daß die Charakterzeile für den VIC erst begann, als wir mit unserer Manipulation aufhörten. Nun arbeiten wir im Prinzip genauso, nur daß wir die Chrakterzeile nicht VOR dem Rasterstrahl wegschieben, sondern MIT ihm. Verschieben wir den Vertikal-Offset nämlich so, daß er immer mit dem Anfang einer Charakterzeile zusammenfällt, so meint der VIC, er müsse jetzt die neuen Charakterdaten lesen, selbst wenn er das in der Rasterzeile zuvor auch schon getan hat. Schalten wir nun gleichzeitig auch noch auf eine andere Video-Map um, so liest der VIC, so als wäre alles in Ordnung, die 40 Charakter der neuen Map!
Bevor ich mich nun aber in theoretischen Erklärungen verliere, möchte Ich Ihnen ein Programmbeispiel zeigen, das alles einfacher zu erklären vermag. Sie finden es auf dieser MD unter dem Namen "GO-FLI-CODE", und müssen es mit der Endung",8,1" laden. Gleichzeitig (und ebenfalls mit ",8,1") sollte auch noch die lange Grafik "GO-FLIPIC" geladen werden, damit Sie beim Start der Routine mittels "SYS4096", auch etwas auf dem Bildschirm sehen. Es handelt sich dabei um das Logo unseres Schwestermagazins "Game On", und zwar in schillernd bunten Farbabstufungen!
Doch kommen wir nun zum Sourcecode dieser FLI-Routine. Zunächst einmal werde ich Ihnen die Initialisierung hier auflisten, die größtenteils identisch mit den zuvorigen Init-

Routinen ist, jedoch auch einige FLI spezifische Einstellungen vornimmt:

```
Init    SEI             ;IRQs sperren
        LDA $7F         ;CIA-IRQs und NMIs
        STA $DC0D       ;sperren
        STA $DD0D
        BIT $DC0D       ;ggf. vorhandene IRQ-
        BIT $DD0D       ;oder NMI-Anfr. löschen
        LDA #$0B        ;Bildschirm
        STA $D011       ;abschalten
        LDY #$00        ;Daten für Color-Map
color   LDA $3C00,Y     ;von $3C00
        STA $D800,Y     ;nach $D800
        LDA $3D00,Y     ;kopieren
        STA $D900,Y
        LDA $3E00,Y
        STA $DA00,Y
        LDA $3F00,Y
        STA $DB00,Y
        INY
        BNE color
        LDA #$00        ;Hardvektor für IRQ bei
        STA $FFFE       ;$FFFE/$FFFF auf
        LDA #$11        ;eigene Routine bei
        STA $FFFF       ;$1100 setzen
        LDX #$38        ;Basisw. Vert.Versch.
        STX $02         ;in Zeropage ablegen
        INX             ;Pro Zeile +1 und
        STX $03         ; ebenfalls in ZP abl.
        INX             ;usw. bis X=$3F
        STX $04
        INX
        STX $05
        INX
        STX $06
        INX
        STX $07
        INX
        STX $08
        STX $09
        LDA #$5C        ;Rasterz. $d012 ist
        STA $d012       ;IRQ-Auslöser
        LDA #$81        ;Raster als IRQ-Quelle
        STA $D01A       ;einschalten
        DEC $D019       ;ggf. VIC-IRQ freigeben
        LDA #$35        ;Basic- u. Betr.Sys.-
        STA $01         :ROM abschalten
        CLI             ;IRQs freigeben
        LDA #$7F        ;Tastaturport initialisieren
        STA $DC00
SPC     LDA $DC01       ;Auf 'SPACE'-Taste
        CMP #$EF        ;warten
        BNE SPC
        LDA #$37        ;ROMs wieder
        STA $01         ;einschalten
        JMP $FEC2       ;und RESET auslösen
```

Hier schalten wir zunächst alle CIA-IRQs ab, tragen die Startadresse unserer FLI-Routine bei $1100 in den Hard-IRQ-Vektor $FFFE/$FFFF ein, schalten das Betriebssystem ab,

damit der Vektor auch angesprungen wird, und erlauben dem VIC in Rasterzeile $5C einen Raster-IRQ auszulösen. Zudem wird eine Tabelle in den Zeropageadressen von $02 bis $09 initialisiert, die in Folge die Werte von $38 bis $3F enthält. Diese müssen wir später, je nach Rasterzeile, in Register $D011 schreiben, damit der Anfang der Charakterzeile immer in der aktuellen Rasterzeile ist. Durch den Basiswert $38 legen wir zunächst nur fest, daß der Bildschirm eingeschaltet ist, und sich der VIC im Grafikmodus und in 25-Charakterzeilen-Darstellung befindet.
Durch das jeweilige aufaddieren von 1 wird dann lediglich der vertikale Verschiebeoffset um 1 erhöht, was dann immer dem jeweiligen Wert für den Beginn einer Charakterzeile entspricht (Startwert $38-> Verschiebung=0 für tatsächliche Charakterzeile; nächster Wert=$39-> Verschiebung=1 Zeile, weshalb die erste Rasterzeile hinter der normalen Charakterzeile vom VIC wieder als Charakterzeile aufgefasst wird; usw.). Die Werte werden übrigens deshalb in eine Tabelle innerhalb der Zeropage eingetragen, damit das Timing später besser klappt.
Des weiteren wird hier der Inhalt der Color-Map initialisiert. Hierzu sollte ich vielleicht noch erläutern, wie das 68 Blocks lange FLIPIC-File aufgebaut ist: zunächst einmal wird es an Adresse $3C00 geladen, wo sich auch die $0400 Bytes für die Color-Map befinden. Ab Adresse $4000 beginnen nun die acht Video-Maps, die später von der FLI-Routine durchgeschaltet werden. Auch sie sind jeweils $0400 Bytes lang (was einer Gesamtlänge von $0400*8=$2000 Bytes gleichkommt). Hiernach, im Bereich von $6000-$8000 befindet sich nun die darzustellende Grafik-Bitmap. Was die Init-Routine nun macht, ist lediglich die Color-Map-Daten zwischen $3C00 und $4000 nach $D800 (Basisadresse des Color-RAMs), zu kopieren. Die Video-RAM-Daten liegen in den Speicherbereichen, die mit den Werten von 0-7 für die Highbits von Register $D018, angewählt werden können (was gleichzeitig auch der Anordnung in Rasterzeilen entspricht - Video-RAM Nr.0 ($4000) für die 0 . Rasterzeile innerhalb Charakterzeile; Video-RAM Nr.1 ($4400) für die 1. Rasterzeile innerhalb der Charakterzeile; usw.).
Kommen wir nun zur Interruptroutine selbst. Sie beginnt wie all unsere zeitkritischen Raster-IRQs zunächst mit der Glättungsroutine, wobei ein weiterer Interrupt festgelegt und ausgelöst wird, bevor die eigentliche FLI-Routine folgt.
Sie beginnt wieder ab dem Label "Onecycle", ab dem der IRQ " geglättet" ist.
Hier nun der eigentliche Kern der Routine. Da der Interrupt bei Rasterzeile $5C ausgelöst wurde, und die Glättung zwei Rasterzeilen verbrauchte, befinden wir uns nun also in Zeile $5E, womit wir also zwei Rasterzeilen vor Beginn der achten Charakterzeile stehen:

```
DEC $D019       ;VIC-ICR löschen
LDA #$5C        ;Neue IRQ-Rasterzeile
STA $D012       ;festlegen ($5C)
LDA #$00        ;IRQ-Vektor wieder auf
STA $FFFE       ;die erste IRQ-Routine
LDA #$11        ;verbiegen (für den
STA $FFFF       ;nächsten Aufruf)
LDA #$00        ;Bildschirmrahmen und
STA $D020       ;Hintergrundfarbe auf
STA $D021       ;'schwarz' setzen
NOP             ;Hier folgen nun 40 NOPs
...             ;um bis zum richtigen
NOP             ;Zeitpunkt zu verzögern
LDY #$08        ;Rasterzeilenzähler initialisieren
JSR fliline     ;8 FLI-Rasterzeilen
JSR fliline     ;für 12 Charakterzeilen
JSR fliline     ;durchführen (gesamte
JSR fliline     ;Größe des FLI-Bereichs:
JSR fliline     ;8x12=96 Rasterzeilen)
JSR fliline
```

```
        JSR fliline
        JSR fliline
        JSR fliline
        JSR fliline
        JSR fliline
        JSR fliline
        LDA #$02        ;Bildschirmrahmenund
        STA $D020       ;Hintergrundfarbe auf
        STA $D021       ;rot' setzen
        LDA #$38        ;Vertikalverschiebung
        STA $D011       ;zurücksetzen (=0)
        LDA #$88        ;VideoMap 8
        STA $D018       ;einschalten
        PLA             ;Prozessorregister vom
        TAY             ;Stapel zurückholen
        PLA             ;und IRQ beenden.
        TAX
        PLA
        RTI
```

Hier werden erst einmal die Vorbereitungen für den nächsten Interrupt getroffen (also für den nächsten Bildaufbau). Dies ist das Rücksetzen des IRQ-Vektors, auf die Glättung-IRQ-Routine, das neue festlegen der Rasterzeile $5C als IRQ-Auslöser, sowie das Löschen des VIC-ICR-Registers durch Zugriff auf Adresse $D019. Nun wird das Y-Register mit dem Wert 8 initialisiert, der in der nun folgenden "FLILINE"-Routine gebraucht wird. Selbige führt den eigentlichen FLI-Effekt aus, wobei Sie immer acht Rasterzeilen lang, seit Beginn einer Rasterzeile, den Effekt erzeugt. Durch den 12-maligen Aufruf lassen wir ihn also ab Rasterzeile $5E für 96 Rasterzeilen aktiv sein. Hier nun der Sourcecode zur FLILINE-Routine:

```
fliline STY $D018       ;Video-Map0 einschalt.
        LDX $02         ;Vert. Versch.=0($38)
        STX $D011       ;schreiben
        LDA #$18        ;VideoMap1-Wert laden
        LDX $03         ;Vert.Versch.=1 ($39)
        NOP             ;Verzögern...
        NOP
        NOP
        NOP
        STA $D018       ;Werte zeitgenau in
        NOP             ;$D018 und $D011
        STX $D011       ;eintragen
        LDA #$28        ;VideoMap2-Wert laden
        LDX $04         ;Ver.Versch.=1 ($3A)
        NOP             ;Verzögern...
        NOP
        NOP
        NOP
        STA $D018       ;Werte zeitgenau in
        NOP             ;$D018 und $D011
        STX $D011       ;eintragen
        LDA #$38        ;Dito für die folgenden
        LDX $05         ;6 Rasterzeilen
        NOP
        NOP
        NOP
        NOP
        STA $D018
```

```
NOP
STX $D011
LDA #$48
LDX $06
NOP
NOP
NOP
NOP
STA $D018
NOP
STX $D011
LDA #$58
LDX $07
NOP
NOP
NOP
NOP
STA $D018
NOP
STX $D011
LDA #$68
LDX $08
NOP
NOP
NOP
NOP
STA $D018
NOP
STX $D011
LDA #$78
LDX $09
NOP
NOP
NOP
NOP
STA $D018
LDY #$08
STX $D011
RTS
```

Dieser etwas merkwürdige Aufbau der Routine ist mal wieder absolut unerläßlich für die richtige Funktionsweise des FLI-Effektes. Wie immer muß hier extrem zeitgenau gearbeitet werden, damit die einzelnen Befehle zum richtigen Zeitpunkt ausgeführt werden. Durch die NOPs innerhalb der eigentlichen IRQ-Routine wird die FLILINE-Routine zum ersten Mal genau dann angesprungen, wenn sich der Rasterstrahl kurz vor dem Beginn einer neuen Charakterzeile befindet. Durch den "STY $D018"- Befehl am Anfang der Routine wird nun gleich auf Video-Map 0 geschaltet, deren Inhalt für die Farbgebung der Grafikpixel in dieser Rasterzeile zuständig ist. Gleichzeitig setzen wir den vertikalen Verschiebeoffset auf Null, wobei mit dem Wert von $38, der sich in der Zeropageadresse $02 befindet, gleichzeitig die Grafikdarstellung eingeschaltet wird. Genau nachdem diese beiden Werte geschrieben wurden beginnt nun die Charakterzeile, in der der VIC, wie wir ja wissen, den Prozessor, und damit unser Programm, für 42 Taktzyklen anhält. Hiernach haben wir noch 21 Taktzyklen Zeit, die Werte für die nächste Rasterzeile einzustellen, wobei auch dies exakt vor Beginn derselben geschehen muß. Hierzu wird zunächst der Akku mit dem Wert $18 geladen (Wert für Register $D018), wobei die oberen vier Bits die Lage der Video-Map bestimmen. Sie enthalten den Wert 1 und bezeichnen damit Video-Map1 bei Adresse $4400 . Die unteren vier Bits bestimmen die Lage des Zeichensatzes

und könnten eigentlich jeden beliebigen Wert enthalten. Da Sie normalerweise den Wert 8 enthalten, benutzen wir ihn ebenfalls. Der LDA-Befehl verbraucht nun 2 Taktzyklen. Als Nächstes wird das X-Register mit dem Wert für Register $D011 inititalisiert. Er ist diesmal $39, was der Vertikal-Verschiebung des Bildschirms um eine Rasterzeile entspricht. Hierbei wird der Wert aus Speicherzelle $03 der Zeropage ausgelesen. Vielleicht wird Ihnen jetzt auch klar, warum wir die Wertetabelle überhaupt, und dann ausgerechnet in der Zeropage angelegt haben: Durch die ZP-Adressierung verbraucht der LDX-Befehl nämlich 3 Taktzyklen, anstelle von nur Zweien (bei direktem Laden mit "LDX #$39"), was für unser Timing besonders wichtig ist! Es folgen nun 4 NOP-Befehle, die einfach nur 8 Taktzyklen verbrauchen sollen, damit wir zum richtigen Zeitpunkt in die VIC-Register schreiben. Was dann auch tatsächlich geschieht, wobei wir mit den Befehlen "STA","STX" und "NOP" nochmals 10 Takte "verbraten" und uns wieder genau am Beginn der nächsten Rasterzeile befinden.
Durch die im letzten Moment vorgegaukelte Vertikal-Verschiebung um eine Rasterzeile, meint der VIC nun, daß er sich wieder am Anfang einer Charakterzeile befindet, weswegen er sie auch prompt einliest. Sinnigerweise jedoch aus der neu eingeschalteten Video-Map Nr.1 !!!
Dies setzt sich so fort, bis auch die letzte der acht Rasterzeilen, nach Beginn der eigentlichen Charakterzeile, abgearbeitet wurde. Nun wird zum Haupt-IRQ zurück verzweigt, wo "FLILINE" sogleich zur Bearbeitung der nächsten Charakterzeile aufgerufen wird.
Wenn Sie übrigens einmal die Summe der verbrauchten Taktzyklen pro Rasterzeile berechnen, so wird Ihnen auffallen, daß wir 23(=2+3+10+8), anstelle von 21 Taktzyklen verbraucht haben, so daß unser Programm also 2 Zyklen länger dauert, als es eigentlich sollte. Dies liegt an einem kleinen "Nebeneffekt" von FLI. Dadurch, daß wir den VIC austricksen, scheint er etwas "verwirrt" zu sein, weswegen er nicht gleich die 40 Zeichen der neuen Video-Map einliest, sondern 24 Pixel lang erst mal gar nicht weiß, was er machen soll (vermutlich liest er sie zu spät, weswegen er nichts darstellen kann) . Aus diesem Grund können auch die ersten 24 Pixel einer FLI-Grafik nicht dargestellt werden. Wenn Sie sich das Beispielbild einmal genauer ansehen, so werden Sie merken, daß es nur 296 Pixel breit ist (3 Charakter fehlen auf der linken Seite) . Erst danach kann der VIC die Grafik mit aktiviertem FLI-Effekt darstellen. In den 3 Charaktern davor ist dann wieder das Bitmuster der letzten Speicherzelle seines Adressierungsbereiches zu sehen (normalerweise $3FFF - im Beispiel aber durch die Bereichsverschiebung $7FFF). Gleichzeitig scheint der Prozessor dadurch aber wieder 2 Zyklen mehr zu haben, was die einzige Erklärung für die oben aufgezeigte Diskrepanz sein kann (Sie sehen: selbst wenn man einen solchen Effekt programmieren kann - bleiben manche Verhaltensweisen der Hardware selbst dem Programmierer ein Rätsel)

4. Weitere Programmbeispiele

Das war es dann wieder einmal für diesen Monat. Ich hoffe, daß Sie meinen, manchmal etwas komplizierten, Ausführungen folgen konnten. Mit der Erfahrung aus den letzten Kursteilen sollte das jedoch kein Problem für Sie gewesen sein. Wenn Sie sich ein paar FLI-Bilder anschauen möchten, so werfen Sie einen Blick auf diese Ausgabe der MD. Außer dem oben schon angesprochenen "GO-FLIPIC", haben wir Ihnen noch zwei weitere FLI-Bilder mit auf die Diskette kopiert."FULL-FLIPIC", ist ein Multicolor-FLI- Bild, daß über den gesamten Bildschirm geht.
Viel mehr als es anzuschauen können Sie nicht damit machen, da sich der Prozessor während des annähernd gesamten Bildaufbaus im FLI-Interrupt befindet, und dadurch wenig Rechenzeit für andere Effekte übrigbleibt. Das dritte Beispiel ist ein FLI-Bild in HIRES-Darstellung ("HIRES-FLIPIC") . Hier sehen Sie wie Eindrucksvoll der FLI-Effekt sein kann, da durch die hohe Auflösung noch bessere Farbeffekte erzielt werden. Zu

jedem der drei Bilder müssen Sie das gleichnamige Programm laden (erkennbar an der Endung "-CODE"), das immer mit "SYS4096" gestartet und durch einen 'SPACE'-Tastendruck beendet wird. Das Prinzip der einzelnen Routinen ist immer ähnlich, nur daß bei "FULL" noch mehr Rasterzeilen in FLI-Darstellung erscheinen, und bei "HIRES" zusätzlich der Hires- Modus zur Darstellung verwendet wird. Am Besten Sie disassemblieren sich die drei Programme mit Hilfe eines Speichermonitors und manipulieren sie ein wenig, um die Vielfalt und Funktionsweise von FLI-Effekten besser zu erlernen.
Übrigens: Der FLI-Effekt funktioniert auch im normalen Textmodus. Hierbei werden dann immer nur die oberen acht Pixel eines Zeichens in allen acht Rasterzeilen wiederholt. Man könnte damit einen recht eindrucksvollen Texteinund ausblend- Effekt programmieren, bei dem die einzelnen Zeichen quasi auf den Bildschirm "fließen" (oder von ihm "wegschmelzen"). Dies als Anregung für ein eigenes FLI-Projekt...

(ih/ub)

Teil 9 – Magic Disk 07/94

Herzlich Willkommen zum neunten Teil unseres Raster-IRQ- Kurses. In dieser Ausgabe soll es um die trickreiche Programmierung von Sprites gehen, die wir bisher ja nur am Rande angeschnitten hatten. Durch Raster-IRQs ist es nämlich möglich, mehr als acht Sprites auf den Bildschirm zu zaubern! Wir wollen hierzu eine sogenannte Sprite-Multiplexer- Routine kennenlernen, mit der wir bis zu 16 Sprites (fast) frei über den Bildschirm bewegen können!

1. Das Funktionsprinzip

Zunächst wollen wir klären, wie man im Allgemeinen die Anzahl der Sprites erhöht. Das Prinzip ist höchst simpel:
dadurch nämlich, daß man einen Rasterinterrupt z. B. in der Mitte des sichtbaren Bildschirms auslöst, hat man die Möglichkeit die Sprite-Register des VIC neu zu beschreiben, um z. B. neue Spritepositionen und Spritepointer zu setzen.
Bevor der Rasterstrahl nun wieder die obere Bildschirmhälfte erreicht, können dann wieder die Daten der ersten acht Sprites in den VIC geschrieben werden.
Auf diese Weise kann man in der oberen, sowie in der unteren Bildschirmhälfte je acht Sprites darstellen. Wie einfach dieses Prinzip funktioniert soll folgendes Programmbeispiel verdeutlichen, das Sie auf dieser MD auch unter dem Namen "16 SPRITES" finden, und wie immer mit ",8,1" laden und durch ein "SYS4096" starten. Nach dem Start werden Sie 16(!) Sprites auf dem Bildschirm sehen, acht in der oberen und acht in der unteren Bildschirmhälfte, jeweils in einer Reihe. Kommen wir zunächst zur Init-Routine unsres kleinen Beispiels, deren Funktionsprinzip uns mittlerweile bekannt sein sollte:

```
Init:   SEI             ;IRQs sperren
        LDA #6          ;Farbe auf 'blau'
        STA $D020
        STA $D021
        LDA #$7F        ;Alle Interruptquellen
        STA $DC0D       ;von IRQ- und NMI-CIA
        STA $DD0D       ;sperren ggf. aufge-
        BIT $DC0D       ;tretene IRQs frei-
        BIT $DD0D       ;geben
        LDA #$94        ;Rasterzeile $94 als
        STA $D012       ;IRQ-Auslöser
        LDA $D011       ;festlegen
        AND #$7F
```

```
                STA $D011
                LDA #$01        ;Rasterstrahl ist
                STA $D01A       ;Interruptquelle
                LDA #<irq1      ;Vektor auf erste IRQ-
                STA $FFFE       ;Routine verbiegen
                LDA #>irq1
                STA $FFFF
                LDY #19         ;Pseudo-Sprite-Register
lo1             LDA vic1,Y      ;in Zeropage von
                STA $80,Y       ;von $80 bis $C0
                LDA vic2,Y      ;kopieren
                STA $A0,Y
                DEY
                BPL lo1
                LDY #62         ;Spriteblock Nr. 13
                LDA #$FF        ;mit $FF auffüllen
lo2             STA 832,Y       ;$0340
                DEY
                BPL lo2
                LDY #7          ;Spritepointer aller
lo3             LDA #13         ;Sprites auf Block 13,
                STA 2040,Y      ;sowie Farbe aller ($07F8)
                LDA #1          ;Sprites auf 'weiß'
                STA $D027,Y     ;setzen
                DEY
                BPL lo3
                LDA #$35        ;ROM abschalten
                STA $01
                CLI             ;IRQs freigeben
lo4             JSR $1200       ;Bewegunsroutine aufrufen
                LDA $DC01       ;SPACE-Taste abfragen
                CMP #$EF
                BNE lo4
                LDA #$37        ;Wenn SPACE, dann ROM
                STA $01         ;wieder einschalten
                JMP $FCE2       ;und RESET auslösen.
```

Hier schalten wir nun also wie gewohnt alle Interruptquellen der CIA ab, und aktivieren den Raster-IRQ, wobei dieser das erste Mal in Rasterzeile $94 auftreten soll, was in etwa die Mitte des sichtbaren Bildbereichs ist. Dort soll dann die Routine "IRQ1" aufgerufen werden, deren Adresse in den Hard-IRQ-Vektor bei $FFFE/$FFFF geschrieben wird. Damit der Prozessor beim Auftreten des IRQs auch tatsächlich unsere Routine anspringt, wird zuvor noch das ROM abgeschaltet, und anschließend in eine Schleife gesprungen, die auf die SPACE-Taste wartet, und in dem Fall einen Reset auslöst. Wichtig sind nun noch die drei Kopierschleifen innerhalb der Initialisierung. Die erste davon ("LO1") kopiert nun zunächst eine Tabelle mit Sprite-Register-Werten, die am Ende des Programms stehen, in die Zeropage ab Adresse $80. Was es damit auf sich hat, sehen wir später. Die zweite und dritte Schleife füllen dann noch den Spriteblock 13 mit gesetzten Pixeln, setzen die Spritepointer aller Sprites auf diesen Block, sowie die Farbe Weiß als Spritefarbe. Sehen wir nun, was die Interruptroutine "IRQ1" macht:

```
IRQ1            PHA             ;Prozessorregister
                TXA             ;retten
                PHA
                TYA
                PHA
                LDA #0          ;Farbe auf 'schwarz'
                STA $D020
```

```
STA $D021
LDA #$FC        ;Rasterzeile $ FC soll
STA $D012       ;nächster IRQ-Auslöser
LDA #< irq2     ;sein, wobei Routine
STA $FFF        ;"IRQ2" angesprungen
LDA #>irq2      ;werden soll
STA $FFFF
DEC $D019       ;VIC-ICR freigeben
LDA $A0         ;X-Pos. Sprite 0
STA $D000       ;setzen
LDA $A1         ;Y-Pos. Sprite 0
STA $D001       ;setzen
LDA $A2         ;Ebenfalls für Sprite 1
STA $D002
LDA $A3
STA $D003
LDA $A4         ;Sprite 2
STA $D004
LDA $A5
STA$D005
LDA $A6         ;Sprite 3
STA $D006
LDA $A7
STA $D007
LDA $A8         ;Sprite 4
STA $D008
LDA $A9
STA $D009
LDA $AA         ;Sprite 5
STA $D00A
LDA $AB
STA $D00B
LDA $AC         ;Sprite 6
STA $D00C
LDA $AD
STA $D00D
LDA $AE         ;Sprite 7
STA $D00E
LDA $AF
STA $D00F
LDA $B0         ;High-Bits der X-Pos.
STA $D010       ;aller Sprites setzen
LDA $B1         ;Sprite-Enable setzen
STA $D015       ;(welche sind an/aus)
LDA $B2         ;X-Expansion setzen
STA $D017
LDA $B3         ;Y-Expansion setzen
STA $D01D
LDA #6          ;Farbe wieder ' blau'
STA $D020
STA $D021
PLA             ;Prozessorregister vom
TAY             ;Stapel zurückholen
PLA
TAX
PLA
RTI             ;und IRQ beenden.
```

Wie Sie sehen, tut die Routine eigentlich nichts anderes, als Werte aus den

Zeropageadressen von $A0 bis $B3 in einzelne VIC-Register zu kopieren. Damit geben wir dem VIC wir ab der Rasterposition $94 also einfach neue Spritewerte.
Gleiches macht nun auch die Routine "IRQ2", die an Rasterzeile $FC ausgelöst wird, nur daß sie die Werte aus den Zeropageadressen von $80 bis $93 in den VIC-Kopiert. In den beiden genannten Zeropage-Bereichen haben wir also quasi eine Kopie der wichtigsten Sprite-Register für jeweils zwei Bildschirmbereiche untergebracht, deren Inhalte jeweils an Rasterzeile $94 und $FC in den VIC übertragen werden. Verändern wir diese Werte nun innerhalb der Zeropage, so können wir jedes der 16 sichtbaren Sprites einzeln bewegen, ein oder ausschalten, sowie X-, bzw. Y-Expandieren.
Wir haben also quasi zwei "virtuelle", oder "softwaremäßige" Sprite-VICs erschaffen, deren Register wie die des normalen VICs beschrieben werden können.
Dies können Sie übrigens mit einer Routine ab Adresse $1200 machen. Sie wird im Hauptrogramm (siehe INIT-Listing) ständig aufgerufen, womit ich Ihnen die Möglichkeit der Spritebewegung offenhalten wollte. Im Beispiel steht an dieser Adresse nur ein "RTS", das Sie jedoch mit einer eigenen Routine ersetzen können.
Wir haben nun also 16 Sprites auf dem Bildschirm, jedoch mit der Einschränkung, daß immer nur jeweils acht im oberen und unteren Bildschirmbereich erscheinen dürfen. Setzen wir die Y-Position eines Sprites aus dem unteren Bereich auf eine Zahl kleiner als $94, so wird es nicht mehr sichtbar sein, da diese Position ja erst dann in den VIC gelangt, wenn der Rasterstrahl schon an ihr vorbei ist. Umgekehrt darf ein Sprite im oberen Bildbereich nie eine Position größer als $94 haben. Außerdem ist noch ein weiterer Nachteil in Kauf zu nehmen: das Umkopieren der Register ist zwar schnell, da wir absichtlich mit Zeropageadressen arbeiten, auf die der Zugriff schneller ist, als auf Low-/ High-Byte-Adressen (2 Taktzyklen, anstelle von 3), jedoch dauert es immer noch knappe 4 Rasterzeilen, in denen gar keine Sprites dargestellt werden können, da es dort zu Problemen kommen kann, wenn der VIC teilweise schon die Werte der oberen und der unteren Sprites enthält.

2. Die Optimierung

Wie Sie in obigem Beispiel sahen, ist die Programmierung von mehr als 8 Sprites recht problemlos, solange alle weiteren Sprites in der Horizontalen voneinander getrennt dargestellt werden können. Was nun aber, wenn Sie z. B. ein Action-Spiel programmieren möchten, in dem mehr als acht Sprites auf dem Bildschirm darstellbar sein sollen, und zudem auch noch möglichst kreuz und quer beweglich sein müssen? Für diesen Fall brauchen wir eine etwas intelligentere Routine, die es uns ermöglicht, flexibler mit den horizontalen Sprite-Positionen umzugehen. Solch eine Routine werden wir nun realisieren. Sie ist allgemeinhin unter dem Namen "Sprite-Multiplexer" bekannt. Wer sich nichts darunter Vorstellen kann, der sollte sich auf dieser MD einmal das Double-Density- Demo anschauen, in dem die Hintergrundsterne, sowie die Meteore, die über den Bildschirm huschen auf diese Art und Weise dargestellt werden.
Kommen wir zunächst zum Grundprinzip der Multiplex-Routine. Mit ihr soll es uns möglich sein,16 Sprites auf dem Bildschirm darzustellen, wobei wir uns möglichst wenig Sorgen über die Darstellung machen wollen. Diese Arbeit soll unsere Routine übernehmen, und automatisch die richtige Darstellung wählen.
Damit es keine Bereiche gibt, in denen gar keine Sprites dargestellt werden können, weil gerade irgendwelche Pseudo-VIC-Daten kopiert werden, sollte Sie zusätzlich auch noch möglichst schnell sein, bzw. den Zeitpunkt der anfallenden Werteänderungen im VIC sorgfältig auswählen können.
Um nun all diesen Anforderungen zu genügen, legen wir uns wieder einen "virtuellen" VIC an, den wir so behandeln, als könne er tatsächlich 16 Sprites darstellen. Seine Register sollen wieder in der Zeropage zu finden sein, damit die Zugriffe darauf schneller ausgeführt werden können. Hierzu belegen wir die obere Hälfte der Zeropage mit den

benötigten Registerfunktionen. Beachten Sie bitte, daß in dem Fall keine Betriebssystemroutinen mehr verwendet werden können, da diese nämlich ihre Parameter in der Zeropage zwischenspeichern und somit unseren VIC verändern würden! Hier nun zunächst eine Registerbelegung unseres Pseudo-VICs:

$80-$8F	16 Bytes, die bestimmen, welche Sprites ein- oder ausgeschaltet sind. Eine 0 in einem dieser Bytes schaltet das entsprechende Sprite aus. Der Wert 1 schaltet es ein ($80 ist für Sprite0,$8 F für Sprite15 zuständig).
$90-$9F	Bytes Diese 16 Bytes halten das Low-Byte der X-Koordinaten der 16 Sprites.
$A0-$AF	In diesen 16 Bytes sind die High-Bytes der X-Koordinaten der 16 Sprites untergebracht.
$B0-$BF	Hier sind nacheinander alle Y-Koordinaten der 16 Sprites zu finden.
$C0-$CF	Diese Register legen die Farben der 16 Sprites fest.
$D0-$DF	Hier werden die Spritepointer untergebracht, die angeben, welcher Grafikblock durch ein Sprite dargestellt wird.
$E0-$EF	Dies ist ein Puffer für die Y-Positionen der 16 Sprites. Er wird für die Darstellung später benötigt.
$F0-$FF	Innerhalb dieses Bereichs werden Zeiger auf die Reihenfolge der Sprites angelegt. Mehr dazu später.

Die hier angegebenen Register können nun von uns genauso verwendet werden, als könne der VIC tatsächlich 16 Sprites darstellen. Möchten wir also z. B. Sprite Nr.9 benutzen, so müssen wir zunächst Xund Y-Position durch Beschreiben der Register $99,$A9, sowie $B9 setzen, Farbe und Spritepointer in $C9 und $D9 unterbringen, und anschließend das Sprite durch Schreiben des Wertes 1 in Register $89 einschalten. Analog wird mit allen anderen Sprites verfahren, wobei die Sprite-Nummer immer der zweiten Ziffer des passenden Registers entspricht.
Wie muß unsere Multiplex-Routine nun vorgehen, um dem VIC tatsächlich 16 unabhängige Sprites zu entlocken? Man greift hier, wie bei allen Raster-IRQ-Programmen, zu einem Trick: Wie wir bisher gesehen hatten, ist die einzige hardwaremäßige Beschränkung, die uns daran hindert, mehr als acht Sprites darzustellen, die X-Koordinate. Es können also immer maximal acht Sprites mit derselben Y-Koordinate nebeneinander stehen. Daran können wir auch mit den besten Rastertricks nichts ändern. In der Horizontalen, können wir aber sehr wohl mehr als acht Sprites darstellen, und das eigentlich beliebig oft (das erste Beispielprogramm hätte auch problemlos 24, oder gar 32 Sprites auf den Bildschirm bringen können) . Rein theoretisch kann ja ein Sprite, das weiter oben am Bildschirm schon benutzt wurde, weiter unten ein weiteres Mal verwendet werden. Einzige Bedingung hierzu ist, daß das zweite, virtuelle, Sprite mindestens 22 Rasterzeilen (die Höhe eines ganzen Sprites plus eine Zeile "Sicherheitsabstand") unterhalb des ersten virtuellen Sprites liegt. Damit diese Bedingung so oft wie nur möglich erfüllt ist, verwendet man ein echtes VIC-Sprite immer zur Darstellung des nten, sowie des <n+8> -ten virtuellen Sprites. Das echte Sprite0 stellt also das virtuelle Sprite0, sowie das virtuelle Sprite8 dar. Da die Y-Koordinaten beider Sprites jedoch beliebig sein können, müssen wir zuvor eine interne Sortierung der Y-Koordinaten vornehmen, so daß der größtmögliche Abstand zwischen den beiden Sprites erreicht wird. Dies sollte unsere Routine in einem Bildschirmbereich tun, in dem keine Sprites angezeigt werden können, also dann, wenn der Rasterstrahl gerade dabei ist, den oberen und unteren Bildschirmrand zu zeichnen (selbst wenn man diesen auch abschalten kann). Nach der Sortierung können nun die Werte der ersten acht virtuellen Sprites im VIC eingetragen werden, wobei gleichzeitig ermittelt wird, in welcher Rasterzeile Sprite0 zu Ende gezeichnet ist. Für diese Rasterzeile wird ein Raster-Interrupt festgelegt, der die Werte für

Sprite8 in den VIC eintragen soll, und zwar in die Register des "echten" Sprite0. Ebenso wird mit den Sprites von 9-15 verfahren. Jedesmal, wenn das korrespondierende Sprite (n-8) zu Ende gezeichnet wurde, muß ein Rasterinterrupt auftreten, der die Werte des neuen Sprites schreibt.
Kommen wir nun jedoch zu der Routine selbst. Sie finden Sie übrigens auch in den beiden Programmbeispielen "MULTI-PLEX1" und "MULTIPLEX2" auf dieser MD.
Beide werden wie üblich mit ",8,1" geladen um mittels "SYS4096" gestartet.
Die Initialisierung möchte ich Ihnen diesmal ersparen, da Sie fast identisch mit der obigen ist. Wichtig ist, daß durch sie zunächst ein Rasterinterrupt an der Rasterposition $F8 ausgelöst wird, an der unsere Multiplex-Routine ihre Organisationsarbeit durchführen soll. Hierbei wird in die folgende IRQ-Routine verzweigt, die in den Programmbeispielen an Adresse $1100 zu finden ist:

```
BORD    PHA             ;Prozessor-Register
        TXA             ;retten
        PHA
        TYA
        PHA
        LDA #$E0        ;Neuen Raster-IRQ
        STA $D012       ;für Zeile $E0
        LDA #$00        ;und Routine "BORD"
        STA $FFFE       ;festlegen
        LDA #$11
        STA $FFFF
        DEC $D019       ;VIC-IRQs freigeben
        JSR PLEX        ;Multiplexen
        JSR SETSPR      ;Sprites setzen
        PLA             ;Prozessorregister vom
        TAY             ;Stapel zurückholen
        PLA
        TAX
        PLA
        RTI             ;und IRQ beenden.
```

Wie Sie sehen, wird hier lediglich der Rasterinterrupt auf Zeile $E0 gesetzt, sowie die Bord-Routine als IRQ-Routine eingestellt. Die eigentliche Arbeit wird von den Routinen "PLEX" und "SETSPR" durchgeführt, wovon wir uns die Erstere nun genauer anschauen möchten. Sie steht ab Adresse $1300:

```
PLEX    CLC             ;Sprites zaehlen, indem
        LDA $80         ;die Inhalte der Ein-/
        ADC $81         ;Ausschaltregister
        ADC $82         ;aller Sprites einfach
        ADC $83         ;im Akku aufaddiert
        ADC $84         ; werden.
        ADC $85
        ADC $86
        ADC $87
        ADC $88
        ADC $89
        ADC $8A
        ADC $8B
        ADC $8C
        ADC $8D
        ADC $8E
        ADC $8F
        STA $7E         ;Anzahl merken
```

```
TAX             ;Aus Tabelle ONTAB
LDA ONTAB,X     ;den VIC-Wert zum Ein-
STA $7F         ;schalten holen und in $7F ablegen
CPX #$00        ;Keine Sprites an?
BNE clry        ;Nein, also weiter
RTS             ;Sonst Prg. beenden
```

Diese Routine ermittelt zunächst einmal, wieviele Sprites überhaupt eingeschaltet sind. Dies tut sie, indem Sie die Inhalte der Einschalt-Register des Pseudo-VICs aufaddiert, wobei das Ergebnis die Anzahl eingeschalteter Sprites ergibt (wenn an, dann Wert=1, sonst 0). Anschließend wird aus der Tabelle "ONTAB" der Wert ausgelesen, der in das VIC-Register zum Einschalten der Sprites kommen muß, um die gefundene Anzahl Sprites zu aktivieren. Die Liste enthält folgende Werte:

$00,$01,$03,$07,$0F,$1F,$3F,$7F
$FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF

Sind nun weniger als acht Sprites eingeschaltet, so wird auch nur diese Anzahl aktiviert werden. Sind es mehr, so müssen immer alle acht eingeschaltet werden. Der so ermittelte Wert wird dann in der Speicherzelle $7F zwischengespeichert. Für den weiteren Verlauf der Routine ist auch die ermittelte Anzahl notwendig, die in der Zeropageadresse $7E untergebracht wird. Zum Schluß prüft die Routine noch, ob überhaupt ein Sprite eingeschaltet werden soll, und kehrt bei einer Anzahl von 0 unverrichteter Dinge zur IRQ-Routine zurück.
Wenn mindestens ein Sprite eingeschaltet ist, so wird die eigentliche Multiplex-Routine aktiv. Sie muß nun die Y-Koordinaten der Sprites sortieren, und die Verteilung der 16 virtuellen Sprites auf die echten VIC-Sprites übernehmen.
Zur Sortierung benötigen wir noch zwei weitere, jeweils 16 Byte große, Felder.
Im ersten, von Zeropageadresse $ E0 bis $EF, wird eine Kopie der Y-Koordinaten angelegt, welche dann sortiert wird. Da die Sortierroutine die Werte der einzelnen Koordinaten manipulieren muß, ist diese Maßnahme notwendig. Desweiteren darf sie die Spritedaten in den Registern von $80 bis $DF nicht verändern, bzw. vertauschen, da ein Programm, das die viruellen VIC-Register mit Daten füttert, ja immer davon ausgehen muß, daß es bei Sprite0 immer das ein und selbe Sprite anspricht, und nicht eines, das von einer Position weiter hinten hierhin sortiert wurde. Deshalb sortiert die Multiplex-Routine nicht die eigentlichen Sprites in die benötigte Reihenfolge, sondern legt eine Tabelle mit Zeigern auf die Reihenfolge der Sprites an. Selbige wird in den Zeropageadressen von $F0 bis $FF abgelegt. Sie enthält nach der Sortierung Werte zwischen 0 und 15, die als Index auf die dem virtuellen Sprite entsprechenden Register dienen.
Zur eigentlichen Sortierung verwenden wir nun einen ganz einfachen Bubblesort-Algorithmus, der so oft über dem zu sortierenden Feld angewandt wird, wie Sprites eingeschaltet sind. Er ermittelt durch eine Reihe von Vergleichen immer den kleinsten Wert innerhalb der Kopie der Y-Positionen, legt ihn in der Zeiger-Liste bei $F0 ab, und setzt die entsprechende Koordinate auf $FF, damit sie im nächsten Sortierdurchlauf den größtmöglichen Wert enthält und somit nicht noch einmal das Minimum sein kann.
Ebenso müssen wir mit den Y-Koordinaten von abgeschalteten Sprites verfahren, die vor dem eigentlichen Sortieren ebenfalls auf $FF gesetzt werden. Dadurch stehen sie immer am Ende der Liste. Nach der Sortierung kann die Routine dann mit Hilfe der Zeiger die Daten zu den Sprites aus den entsprechenden Registern holen, und in den VIC schreiben.

Bevor wir nun zur Beschreibung der Routine kommen, noch einige technische Hinweise: damit die Routine möglichst schnell arbeitet, wurde auf die Verwendung von Schleifen so weit wie möglich verzichtet. Das heißt, daß z. B jeder der Y-Werte über einen eigenen

CMP-Befehl verfügt, der ihn mit dem aktuellen Minimum vergleicht. Analog ist es bei anderen Vorgängen (so auch bei der Aufaddierung der Sprite-Einschalt-Register oben, wo eigentlich auch eine Schleife hätte benutzt werden können). Dadurch verlängert sich das Programm natürlich ein wenig, jedoch ist der damit verbundene Speicheraufwand noch erträglich und wir erreichen zudem eine hohe Verarbeitungsgeschwindigkeit. Da sich viele Vorgänge oft wiederholen, werde ich an diesen Stellen mit "..." eine Folge andeuten, die analog auch für alle weiteren Sprites ausgeführt wird.
Kommen wir nun also zur eigentlichen Multiplex-Routine, die mit dem Label "CLRY" beginnt, wohin auch die letzte Routine verzweigt, wenn Sprites eingeschaltet sind. Hier wird nun geprüft, ob ein Sprite eingeschaltet ist, oder nicht, und in letzterem Fall die Y-Koordinate mit dem Wert $FF überschrieben, damit das abgeschaltete Sprite bei der Sortierung später nicht mehr berücksichtigt wird:

```
CLRY    LDY #$FF        ;Y-Reg mit $FF laden
sx0     LDA $80         ;Sprite0 an?
        BNE sx1         ;Ja, also weiter
        STY $B0         ;Nein, also $FF in Y-Pos
sx1     LDA $81         ;Sprite 1 an?
        BNE sx2         ;Ja, also weiter
        STY $B1         ;Nein, also $FF in Y-Pos
sx2     …               ;Analog für Sprites 3-14 sx15
        LDA $8F         ;Sprite 15 an?
        BNE YCOPY       ;Ja, also zu YCOPY springen
        STY $BF         ;Nein, also $ FF in Y-Pos
```

Damit hätten wir nun also alle Y-Positionen abgeschalteter Sprites gelöscht.
Da die Routine die Y-Positionen der eingeschalteten Sprites nicht verändern darf, wird nun eine Kopie dieser Werte in $E0 bis $EF angelegt:

```
YCOPY   LDA $B0         ;Y-Wert Sprite 0
        STA $E0         ;kopieren
        …               ;Analog für Sprites 1-14
        LDA $BF         ;Y-Wert Sprite 15
        STA $EF         ;kopieren
```

Nachdem nun auch das Sortierfeld angelegt wurde, können wir endlich mit der Sortierung beginnen. Hierbei benutzen wir das Y-Register um den momentan kleinsten Y-Wert zu speichern. Der Akku wird beim Auffinden eines minimalen Wertes dann immer mit einem Zeiger auf das entsprechende Sprite geladen, der der Spritenummer entspricht. Das X-Register enthält den aktuellen Index auf die Zeigertabelle und wird pro Durchlauf um eins erhöht. Die Sortierung ist beendet, wenn die Vergleichsroutine insgesamt so oft durchlaufen wurde, wie Sprites eingeschaltet sind:

```
SORT    LDX #$00        ;Index initialisieren
Loop    LDY #$FF        ;Y-Min. initialisieren
        CPY $E0         ;YSpr0 < YMin?
        BCC s1          ;Nein, also weiter
        LDY $E0         ;Ja, also YMin=YSpr0
        LDA #$00        ;Zeiger Spr0=0 laden
s1      CPY $E1         ;YSpr1 < YMin?
        BCC s2          ;Nein, also weiter
        LDY $E1         ;Ja, also YMin=YSpr1
        LDA #$01        ;Zeiger Spr1=1 laden
s2      …               ;Analog für Sprites 3-14
s15     CPY $EF         ;YSpr15 < YMin?
        BCC s16         ;Nein, also weiter
```

```
                LDY $EF         ;Ja, also YMin=YSpr15
                LDA #$0F        ;Zeiger Spr15=15
s16             STA $F0,X       ;Zeiger ablegen
                TAY             ;Zgr. als Index in Y-Reg
                LDA  #$FF       ;YSpr mit YMin auf $FF
                STA $E0,Y       ;setzen
                INX             ;Zeiger-Index+1
                CPX $7E         ;mit Anzahl Sprites vergleichen
                BEQ end         ;Gleich, also Ende
                JMP loop        ;Sonst nochmal sortieren
end             RTS
```

Wie Sie sehen, wird nach jedem ermittelten Minimum der entsprechende Y-Wert auf $FF gesetzt, damit er im nächsten Vergleich nicht mehr herangezogen wird. Auf diese Weise wird nun nach und nach immer wieder der kleinste Wert ermittelt, solange, bis der Puffer nur noch $FF-Werte enthält, und die Schleife "Anzahl-Sprites"- Mal durchlaufen wurde. Die Sortierung ist damit beendet, und die Multiplex- Routine kehrt wieder zur IRQ-Routine zurück.
Hier nun wird als Nächstes die "SETSPR"- Routine aufgerufen, die anhand der ermittelten Werte zunächst die Daten der ersten acht virtuellen Sprites in den VIC überträgt. Gleichzeitig berechnet sie mit Hilfe der Y-Position dieser Sprites die Rasterzeile, an der ein IRQ ausgelöst werden muß, um das jeweils achte Sprite nach dem aktuellen anzuzeigen, und setzt den nächsten IRQ-Auslöser entsprechend. Zunächst einmal wollen wir uns den ersten Teil dieser Routine anschauen. Er beginnt ab Adresse $1500:

```
SETSPR          LDA $7F         ;VIC-Wert für eingesch.
                STA $D015       ;Sprites setzen
                LDA #$00        ;High-Bits der X-Pos
                STA $D010       ;löschen
                LDA $7E         ;Anzahl Sprites holen
                CMP #$01        ;Wenn mind. 1 Spr. ein-
                BCS spr00       ;gesch., dann weiter
                RTS             ;Sonst Ende
spr00           CLC             ;C-Bit für Addition löschen
                LDX $E0         ;Zgr. aus Tabelle holen
                LDA $90,X       ;X-Pos. holen und für
                STA $D000       ;VIC-Spr.0 setzen
                LDA $B0,X       ;Y-Pos. holen und für
                STA $D001       ;VIC-Spr.0 setzen
                ADC #22         ;Raster für Spr.Ende=
                STA ras8+1      ;YPos+22 setzen
                LDA $D0,X       ;Spr.Zeiger holen und
                STA $07F8       ;für VIC-Spr.0 setzen
                LDA $C0,X       ;Spr.Farbe holen und
                STA $D027       ;für VIC-Spr.0 setzen
                LDY $A0,X       ;X-Pos-High holen,
                LDA $D010       ;X-High-Bit-Reg. holen
                ORA high0,Y     ;Wert f. VIC-Spr.0 ein-
                STA $D010       ;odern und zurückschreiben
                LDA $7E         ;Anzahl holen
                CMP #$02        ;Mehr als 1 Spr. an?
                BCS spr01       ;Ja, also weiter.
                RTS             ;Sonst Ende
spr01           ...             :Analog für Sprite 1-7
spr07:          ...             ;Werte f. Spr.7 setzen
                LDA $7E         ;Falls mehr als acht
                CMP #$09        ;Sprites, dann
                BCS acht        ;neuen IRQ setzen
```

```
                RTS             ;Sonst Ende
acht            lda #< spr08    ; Adr. Routine "Spr8"
                STA $FFFE       ;in IRQ-Vektor bei
                LDA #> spr08    ;$ FFFE/$ FFF ein
                STA $FFFF       ;tragen
ras8            LDA #$00        ;Rasterz. Spr0+22 als
                STA $D012       ;IRQ-Quelle setzen
                DEC $D019       ;VIC-IRQs freigeben
                RTS             ;Ende
```

Wie Sie sehen, holt die SETSPR-Routine nun nacheinander alle Zeiger aus der sortierten Tabelle bei $F0 und überträgt die Werte der ersten acht Sprites in den VIC. Auf die Register des Pseudo-VICs wird dabei über X-Register- indizierte Adressierung zugegriffen.
Zwei Dinge sollten nun noch erläutert werden: Zum Einen wird nach Setzen der Y-Position eines Sprites die Rasterzeile berechnet, an der es zu Ende gezeichnet ist. Der so ermittelte Wert wird nun an dem Label "RAS8" plus 1 eingetragen, womit wir den Operanden des LDA-Befehls am Ende der Routine modifizieren. Er lädt nun den Akku mit der besagten Rasterzeilennummer und schreibt ihn in das Raster-IRQ-Register, womit der VIC nachdem er Sprite0 auf dem Bildschirm dargestellt hat, einen Raster-IRQ erzeugt. Hierbei wird dann zur Routine "SPR8" verzweigt, die ich Ihnen gleich erläutern werde.
Analog wird mit den Sprites von 1-7 verfahren, wobei nach jedem Sprite geprüft wird, ob noch ein weiteres Sprite eingeschaltet ist, und demnach initialisiert werden muß. Ist das nicht der Fall, so wird direkt zum IRQ zurückgekehrt. Dadurch wird ebenfalls nur dann ein IRQ auf die Routine "SPR8" eingestellt, wenn tatsächlich mehr als acht virtuelle Sprites eingeschaltet sind. Im anderen Fall brauchen wir ja keine Manipulation vorzunehmen, weswegen der nächste Raster-IRQ, wieder auf "BORD" springt, wo wir die Multiplex-Routine ein weiteres Mal durchlaufen. Sollen nun aber mehr als acht Sprites dargestellt werden, so wird ein IRQ erzeugt, der auf die Routine "SPR08" springt, die wir uns gleich näher anschauen werden.
Die zweite, etwas undurchsichtige Stelle ist das Setzen der High-Bits für die X-Position eines Sprites. Hier gehen wir wie folgt vor: Zunächst wird der High-Wert der X-Position geholt, der nur 0 oder 1 sein, je nach dem ob die X-Position kleiner oder größer/ gleich 256 ist. Dieser Wert wird nun als Zeiger auf eine Tabelle mit X-High- Bit-Werten für das jeweilige Sprite benutzt. Sie steht ganz am Ende des Programms und sieht folgendermaßen aus:

```
high0:          $00,$01
high1:          $00,$02
high2:          $00,$04
high3:          $00,$08
high4:          $00,$10
high5:          $00,$20
high6:          $00,$40
high7:          $00,$80
```

Wie Sie sehen, enthält sie für jedes Sprite einmal ein Nullbyte, das durch die SETSPR-Routine geladen wird, wenn der X-High- Wert Null ist, sowie einen Wert, in dem das Bit gesetzt ist, das im X-High- Bit-Register für das entsprechende Sprite zuständig ist. Durch das Einodern des ermittelten Wertes in dieses Register setzen wir nun letztendlich das High-Bit der X-Position eines Sprites.
Hierbei wird bei den Sprites von 1-7 jeweils auf ein eigenes "High"-Label zugegriffen, bei Sprite 1 also auf "High1", bei Sprite 2 auf "High2" und so weiter.
Kommen wir nun jedoch zur "SPR08"- Routine, die als IRQ-Routine aufgerufen wird, und zwar nachdem Sprite0 auf dem Bildschirm dargestellt wurde:

```
SPR08       PHA             ;Prozessor-Register. retten
            TXA
            PHA
            TYA
            PHA
            LDX $F8         ;Zgr. aus Sort-Liste
            LDA $90,X       ;X-Pos. in VIC-Spr0
            STA $D000       ;eintragen
            LDA $B0,X       ;Y-Pos. in VIC-Spr0
            STA $D001       ;eintragen
            LDA $D0,X       ;Spr.Zgr. in VIC-Spr0
            STA $07F8       ;eintragen
            LDA $C0,X       ;Spr.Farbe in VIC-Spr0
            STA $D027       ;eintragen
            LDY $90,X       ;X-High-Wert holen
            LDA $D010       ;VIC-High-Reg. lesen
            AND #$FE        ;Bit für Spr0 ausmaskieren
            ORA high0,Y     ;Mit Wert f. X-High
            STA $D010       ;odern und zurückschreiben
            LDA #<spr09     ;Adresse f. Raster-IRQ
            STA $FFFE       ;des nächsten Sprite
            LDA #>spr09     ;in IRQ-Vektoren
            STA $FFFF       ;schreiben
            DEC $D019       ;VIC-IRQs freigeben
            LDA $7E         ;Anzahl holen,
            CMP #$0A        ;Spr9 benutzt?
            BCS ras9        ;Ja, also weiter
            JMP bordirq     ;Sonst IRQ rücksetzen
ras9        LDA #$00        ;Rasterz. für Spr9 laden
            CMP $D012       ;mit aktueller. Strahlposition vergleichen
            BMI direct9     ;Wenn größer oder gleich
            BEQ direct9     ;Spr.9 sofort zeichnen
            STA $D012       ;Sonst n. IRQ festlegen
            PLA             ;Prozessor-Regs zurück-
            TAY             ;holen und IRQ
            PLA             ;beenden
            TAX
            PLA
            RTI
```

Wie Sie sehen, werden zunächst wieder wie schon bei den anderen Sprite-Routinen, die virtuellen VIC-Werte in den echten VIC übertragen. Hiernach wird verglichen, ob mehr als neun Sprites dargestellt werden sollen, und wenn ja zur Routine "RAS9" verzweigt, die den IRQ für Sprite9 vorbeitet. An diesem Label befindet sich wieder der LDA-Befehl, der von der Darstellungsroutine für Sprite1 so abgeändert wurde, daß er die Rasterzeile des Endes dieses Sprites in den Akku lädt. Bevor nun der Interrupt gesetzt wird, prüft das Programm durch einen Vergleich des Akkuinhalts mit der aktuellen Rasterstrahlposition, ob der Rasterstrahl an besagter Zeile nicht schon vorüber ist. In dem Fall wird kein IRQ vorbereitet, sondern direkt auf die Routine zur Darstellung des nächsten Sprites verzweigt (siehe BEQ bzw. BMI-Befehle). Diese beginnt dann folgendermaßen:

```
SPR09       PHA             ;Prozessor-Register
            TXA             ;retten (dieser Teil
            PHA             ;wird bei einem IRQ
            TYA             ;abgearbeitet!)
            PHA
direct9     LDX $F9         ;Setzen der Werte für
            LDA $99,X       ;Spr9 in VIC-Spr1
```

...

Wie Sie sehen, wird durch den Sprung auf "direct9" lediglich der IRQ-Teil der Routine übersprungen. Ansonsten ist die Routine identisch mit "SPR8". Ebenso existieren jeweils eigene Routinen für die Sprites von 9 bis 15. Sollten weniger als 15 Sprites dargestellt werden, so wird, wie bei SPR8 auch schon ersichtlich, auf die Routine "BORDIRQ" gesprungen. Sie steht ganz am Ende des Programms und setzt "BORD" wieder als nächste IRQ-Routine und beendet den aktuellen IRQ. Sie wird ebenfalls abgearbeitet, wenn das fünfzehnte, virtuelle Sprite initialisiert wurde, so daß der gesamte Multiplex-Vorgang noch einmal von vorne beginnen kann.
Dies wäre es dann wieder einmal für diesen Monat. Ich möchte Sie noch dazu animieren, ein wenig mit den Multiplex-Routinen herumzuexperimentieren. Versuchen Sie doch einmal 24 Sprites auf den Bildschirm zu bringen! Im übrigen sind nicht immer alle 16 Sprites auf dem Bildschirm darstellbar. Ist der Abstand zwischen zwei virtuellen Sprites, die durch das selbe " echte" Sprite dargestellt werden, kleiner als 22 Rasterzeilen, so erscheint das zweite Sprite nicht auf dem Bildschirm. Des weiteren ist aus dem Multiplex-Demo2 ersichtlich, daß es auch Probleme mit der Sprite-Priorität gibt. Dadurch, daß die Sprites umsortiert werden müssen, kann es passieren, daß manche zunächst Sprites vor, und beim nächsten Bildaufbau hinter anderen Sprites liegen. Da man die Sprite-Priorität nicht beeinflussen kann, und Sprites mit kleinen Spritenummern Prioriät vor Sprites mit großen Spritenummern haben, kommt es hier zu kleinen Darstellungsfehlern, die leider nicht behoben, sondern lediglich unsichtbar gemacht werden können, indem man allen Sprites dieselbe Farbe gibt. In einem Spiel, in dem es wichtig ist, daß die Spielfigur immer vor allen anderen Objekten zu sehen ist, kann man aber auch nur die Sprites von 1-7 multiplexen, und Sprite 0 unberührt von den restlichen lassen. Die Routine kann dann zwar nur 15 Sprites darstellen, korrigiert jedoch den obigen Fehler. Wie Sie sehen ist auch dieser Raster-Effekt in vielfachen anwendungsspezifischen Kombinationsmöglichkeiten umsetzbar.

(ub)

Teil 10 – Magic Disk 08/94

Herzlich Willkommen zum zehnten Teil unseres Raster-IRQ- Kurses. In dieser Ausgabe wollen wir uns weiterhin mit der trickreichen Sprite-Programmierung befassen. Im letzten Kursteil hatten Sie ja schon gelernt, wie man mit Hilfe eines Sprite-Multiplexers mehr als 8 Sprites auf den Bildschirm zaubert. Mit einem ähnlichen Trick werden wir heute einen sogenannten "Movie-Scroller" programmieren, der es uns ermöglicht, einen Scrolltext, so wie in Abspännen von Filmen, von unten nach oben über den gesamten Bildschirm zu scrollen. Hierbei werden wir wieder durch den Multiplex-Effekt insgesamt 104 (!) Sprites aus dem VIC locken!

1. Das Prinzip des Moviescrollers

Das Funktionsprinzip des Movie-Scrollers ist recht einfach und sollte nach Kenntnis der Multiplex-Routinen kein Problem für Sie sein: Jede Zeile unseres Movie-Scrollers soll aus 8 Sprites aufgebaut sein, in denen wir einen Text darstellen. Um nun mehrere Zeilen zu generieren, müssen wir in regelmäßigen Abständen einen Raster-IRQ erzeugen, der jedesmal die Y-Position, sowie die Sprite-Pointer der acht Sprites neu setzt, um somit die nächste Scroller-Zeile darzustellen. Unsere Routine tut dies alle 24 Rasterzeilen, womit sich zwischen zwei Spritezeilen immer 3 Rasterzeilen Freiraum befinden. Um nun einen Scrolleffekt nach oben zu erzeugen, benutzen wir zusätzlich ein Zählregister, daß einmal pro Rasterdurchlauf um 1 erniedrigt, und so von $17 bis $00 heruntergezählt wird. Es dient als zusätzlicher Offset auf die Rasterzeilen, in denen ein IRQ ausgelöst werden muß. Ist

dieses Zählregister auf 0 heruntergezählt, so wird es wieder auf $17 zurückgesetzt und die Spritepointer für jede einzelne Spritezeile werden alle um je eine Zeile höher kopiert. In die, am Ende der Pointerliste, freigewordenen Sprites werden dann die Buchstaben der neuen Zeile einkopiert, die sich dann von unten wieder in den Scroller einreihen können.

2. Der Programmablauf

Um den Movie-Scroll-Effekt besser zu erläutern haben wir natürlich wieder einige Beispielprogramme für Sie parat.
Sie heißen "MOVIE.1" und "MOVIE.2" und befinden sich ebenfalls auf dieser MD.
Wie immer müssen beide Beispiele mit ",8,1" geladen und durch ein "SYS4096" gestartet werden."MOVIE.2" unterscheidet sich von "MOVIE.1" nur darin, daß zusätzlich zum Scroller auch der obere und untere Bildschirmrand geöffnet wurden.
Dies ist nur durch einen ganz besonderen Trick möglich, den wir später noch erläutern werden. Ich möchte Ihnen nun zunächst eine Speicheraufteilung der beiden Routinen geben, damit Sie sich die einzelnen Unterroutinen, die ich nicht alle in diesem Kurs erläutern werde, mit Hilfe eines Disassemblers einmal selbst anschauen können:

Adresse	**Funktion**
$0800	Zeichensatz $1000 IRQ-Init, incl. aller Sprite-Initialiserungen $1100 Movie-IRQ- Routine. Dies ist die eigentliche IRQ-Routine, die alle 24 Rasterzeilen die Sprites neu setzt.
$1200	BORDERNMI (nur in "MOVIE.2"), zum Öffnen des oberen und unteren Bildschirmrandes.
$1300	MOVESPR-Routine. Sie bewegt die 13 Spritezeilen pro Rasterdurlauf um eine Rasterzeile nach oben.
$1400	MAKETEXT-Routine. Diese Routine schreibt den Text in die acht Sprites, die sich gerade am unteren Bildschirmrand befinden.
$1500	Spritepointer-Liste, die auf die Sprites im Bereich von $2600 - $3FFF zeigt.
$1600	Der Scroll-Text im ASCII-Format.

Die Initialisierungsroutine unseres Movie-IRQs schaltet wie üblich das Betriebssystem-ROM ab, und setzt im Hardware- IRQ-Zeiger bei $FFFE/$FFFF die Startadresse der MOVIEIRQ-Routine ($1200) ein. Zudem werden alle CIA-Interrupts gesperrt und die Spriteregister des VICs initialisiert. Hierbei tragen wir lediglich alle X-Positionen der Sprites ein, die bei $58 beginnen, und in 24-Pixel-Schritten pro Sprite erhöht werden. Natürlich muß auch das X-Position-High-Bit des achten Sprites, daß sich ganz rechts auf dem Bildschirm befindet, auf 1 gesetzt werden. Des weiteren wird die Farbe aller Sprites auf "weiß" geschaltet. Zusätzlich wird in einer eigenen Unterroutine der Speicherbereich von $2600-$3FFF, in dem die 104 Sprites unterbringen, gelöscht. Zuletzt legt die Init-Routine Rasterzeile $17 als ersten IRQ-Auslöser fest und erlaubt dem VIC IRQs zu erzeugen.
Die MOVIEIRQ-Routine stellt nun den Kern unseres Movie-Scrollers dar. Es handelt sich hierbei um die IRQ-Routine, die alle 24 Rasterzeilen die Y-Positionen der Sprites ändert. Sie wird zum ersten Mal an Rasterzeile $17 angesprungen und setzt dann die folgenden IRQ-Rasterzeilen von selbst. Hier nun der kommentierte Sourcecode:

```
MOVIEIRQ    PHA             ;Prozessorregister retten
            TXA
            PHA
            TYA
            PHA
            LDA#$FF         ;Alle Sprites
            STA $D015       ;einschalten
            INC $D020       ;Rahmenfarbe erhöhen
```

Nach Einsprung in die IRQ-Routine werden, nach dem obligatorischen Retten aller

Prozessorregister, zunächst alle Sprites eingeschaltet. Anschließend erhöhen wir die Rahmenfarbe um 1, damit Sie sehen können, wie lange es dauert, bis alle Sprites neu gesetzt wurden. Nun beginnt der eigentliche Hauptteil der Routine:

```
CLC             ;C-Bit für Addition löschen
LDY counter     ;Rasterzähler als Y-Index
LDA softroll    ;Akt. Scrolloffs. Holen.
ADC ypos,Y      ;Y-Wert addieren.
STA $D001       ;und selbigen in alle
STA $D003       ;acht Y-Positionen
STA $D005       ;der Sprites eintragen
STA $D007
STA $D009
STA $D00b
STA $D00d
STA $D00f
```

Wir setzen hier die Y-Positionen der Sprites. Hierbei hilft uns eine Tabelle namens "YPOS", in der alle Basis-Y- Positionen der insgesamt 13 Spritezeilen untergebracht sind. Die Y-Positionen der Sprites in der ersten Zeile sind dabei auf den Wert 26 festgelegt. Alle weiteren Positionen resultieren dann aus dem jeweils letzten Positionswert plus dem Offset 24.
Des weiteren erscheinen in diesem Programmteil noch zwei Labels, mit den Namen "SOFTROLL" und "COUNTER". Sie stehen für die Zeropageadressen $F6 und $F7, in denen wir Zwischenwerte unterbringen.
"SOFTROLL" ($F6) ist der oben schon erwähnte Rasterzeilenzähler, der von $17 auf 0 heruntergezählt wird. In "COUNTER" ist vermerkt, wie oft die IRQ-Routine während des aktuellen Rasterdurchlaufs schon aufgerufen wurde. Dies dient gleichzeitig als Zähler dafür, welche Spritezeile wir momentan zu bearbeiten haben. Beim ersten Aufruf enthält "COUNTER" den Wert 0. Auf diese Weise können wir ihn als Index auf die YPOS-Tabelle verwenden. Nachdem der Akku nun also mit den Scrolloffset "SOFTROLL" geladen wurde, kann so der Basis-Y-Wert der entsprechenden Spritezeile (im ersten Durchlauf Zeile 0, Y-Pos 26) auf den Akkuinhalt aufaddiert werden. Der resultierende Wert entspricht nun der Y-Position aller Sprites dieser Zeile, die wir sogleich in die VIC-Register eintragen.
Nun müssen noch die Spritepointer der neuen Spritezeile neu gesetzt werden, da diese ja einen anderen Text enthält als die vorherige:

```
LDY isline      ;Zgr.- Index in Y holen
LDX pointer,Y   ;Zgr.-Basiswert aus Tabelle
STX $07F8       ;für Sprite0 setzen
INX             ;um 1 erhöhen und
STX $07F9       ;für Sprite1 setzen
INX             ;Ebenso für Sprites2-7
STX $07FA
INX
STX $07FB
INX
STX $07FC
INX
STX $07FD
INX
STX $07FE
INX
STX $07FF
```

Auch hier verwenden wir ein Label um auf eine Zeropageadresse zuzugreifen."ISLINE"

steht für Adresse $F9, die uns als Zwischenspeicher für einen Index auf die Spritezeigerliste dient. In letzterer sind nun alle Spritepointerwerte für das jeweils 0. Sprite einer jeden Zeile untergebracht. Die Zeiger für die Sprites von 1 bis 7 resultieren aus dem aufaddieren von 1 auf den jeweils letzten Wert, was in unserer Routine durch die INX-Befehle durchgeführt wird. Die Zeigertabelle enthählt nun die Werte von $98 bis $F8, als Zeiger auf die Sprites, die im Speicherbereich von $2600-$3FFF liegen, jeweils in Achterschritten. Hier eine Auflistung der kompletten Tabelle:

```
pointer     .byte $98,$A0,$A8,$B0
            .byte $B8,$C0,$C8,$D0
            .byte $D8,$E0,$E8,$F0,F8
            .byte $98,$A0,$A8,$B0
            .byte $B8,$C0,$C8,$D0
            .byte $D8,$E0,$E8,$F0,F8
```

Wie Sie sehen, liegen hier die angesprochenen Pointer-Werte zweimal vor. Das ist notwendig, um das Umschalten der Spritezeilen, wenn diese aus dem oberen Bildschirmrand herausgescrollt werden, zu vereinfachen. Verschwindet nämlich die erste Spritezeile, deren Spritepointer von $98-$9F gehen, womit ihre Sprites von im Bereich von $2600 bis $2800 untergebracht sind, aus dem oberen Bildschirmrand, so muß die zweite Spritezeile (Pointer von $A0 - $A7) von nun an die erste, auf dem Bildschirm darzustellende, Zeile sein, wobei wir die gerade herausgescrollte Zeile als unterste Zeile verwenden müssen. Nachdem ihr Textinhalt in die Sprites im Speicherbereich von $2600 bis $2800 eingetragen wurde, müssen nur noch die Spritezeiger zum richtigen Zeitpunkt auf diese Zeile umgeschaltet werden. Wir haben nun noch einen Weiteren Index, namens "SHOWLINE", der in Zeropageadresse $F8 untergebracht ist, und uns angibt, welche der Spritezeilen als Erstes auf dem Bildschirm dargestellt werden muß. Zu Beginn eines neuen Rasterdurchlaufs wird "ISLINE" mit diesem Index initialisiert. Dadurch, daß unsere Tabelle nun nach dem Wert $F8 ein zweites Mal von vorne beginnt, kann "ISLINE" während des Programmablaufs problemlos inkrementiert werden, ohne dabei einen Zeilenüberlauf beachten zu müssen!
Kommen wir jedoch wieder zurück zu unserer IRQ-Routine. Nachdem die Y-Positionen, sowie die Spritezeiger gesetzt wurden müssen noch einige verwaltungstechnische Aufgaben durchgeführt werden:

```
CLC                 ;C-Bit für Addition löschen
LDY counter         ;Spr-Zeilen- Index holen
LDA ad012+1,Y;      LOW-Byte für. nächsten. IRQ
ADC softroll        ;Scroll-Offs. Add.
STA $D012           ;und als nächsten IRQ setzen
ROR                 ;C-Bit in Akku rotieren
AND #$80            ;und isolieren
ORA ad011+1,Y       ;HIGH-Bit für nächsten IRQ
ORA $D011           ;sowie $D011 einodern
STA $D011           ;und setzen
DEC $D019           ;VIC-IRQs freigeben
DEC $D020           ;Rahmenfarbe zurücksetzen
```

In diesem Teil der Routine wird nun der folgende Raster-IRQ vorbereitet. Hierzu muß zunächst die Rasterzeile ermittelt werden, in der dieser auftreten soll.
Dafür existieren zwei weitere Tabellen, die die Basis-Rasterstrahlpositionen für die 13 Interrupts enthalten. Hierbei wird es wieder etwas kompliziert, da nämlich auch IRQs an Strahlpositionen größer als $FF ausgelöst werden müssen, und wir deshalb das High-Bit der auslösenden Rasterstrahlposition, das in Bit 7 von $D011 eingetragen werden muß,

mitberücksichtigen müssen. Auch hierfür müssen wir sehr trickreich vorgehen:
Zunächst einmal ermitteln wir das Low-Byte der nächsten Rasterposition, indem wir es, mit "COUNTER" als Index im Y-Register, aus der Tabelle "AD012" auslesen. Auf diesen Wert muß nun noch der momentane Scrolloffset aus "SOFTROLL" addiert werden, um die tatsächliche Rasterzeile zu erhalten. Der daraus resultierende Wert kann zunächst einmal in $D012 eingetragen werden. Sollte bei der Addition ein Überlauf stattgefunden haben, also ein Wert größer $FF herausgekommen sein, so wurde dies im Carry-Flag vermerkt. Selbiges rotieren wir mit dem ROR-Befehl in den Akku hinein, und zwar an Bitposition 7, wo auch das High-Bit der IRQ-Rasterstrahls in $D011 untergebracht wird. Nachdem nun dieses High-Bit durch ein "AND $80" isoliert wurde, oder wir aus der Tabelle "AD011" das High-Bit der Standard-Rasterposition (ohne Offset) in den Akku ein. Danach müssen natürlich auch noch alle weiteren Bits von $D011 in den Akku eingeknüpft werden, damit wir beim folgenden Schreibzugriff keine Einstellungen ändern.
Nun muß nur noch das ICR des VIC gelöscht werden, damit der nächste IRQ auch auftreten kann. Gleichzeitig wird die Rahmenfarbe wieder heruntergezählt (dadurch entstehen die grauen Rechen- zeit-Anzeigen im Bildschirmrahmen).
Nun noch der letzte Teil der IRQ-Routine, der prüft, ob schon alle Spritezeilen aufgebaut wurden:

```
                INC isline          ;Akt. Zgr.-Zähler. +1
                INC counter         ;Y-Pos-Zähler +1
                LDA counter         ;Y-Pos-Zähler holen
                CMP #$0C            ;und mit 14 vergleichen
                BNE endirq          ;Wenn ungleich, dann weiter
                JSR movespr
                LDA #$00
                STA $D015
mt              JSR maketext
ENDIRQ          PLA
                TAY
                PLA
                TAX
                PLA
                RTI
```

Hier werden jetzt die Zähler und Indizes für den nächsten IRQ voreingestellt, sowie geprüft, ob schon alle Spritezeilen dargestellt wurden. Ist dies nicht der Fall, so wird der IRQ durch Zurückholen der Prozessorregister, gefolgt von einem RTI, beendet. Befinden wir uns allerdings schon im letzten der 13 IRQs, die pro Rasterdurchlauf auftreten sollen, so fällt der Vergleich von "COUNTER" mit dem Wert 14 negativ aus, womit die Routine "MOVESPR" angesprungen wird.
Sie sorgt für das korrekten Herabzählen des Scrolloffsets, und erkennt, wenn die oberste Spritezeile gerade aus dem Bildschirm gescrollt wurde:

```
MOVESPR         LDA #$00            ;IRQ-Zähler initialisieren
                STA counter
                SEC                 ;C-Bit für Subtraktion setzen
                LDA softroll        ;Scroll-Offs. holen
                SBC #$01            ;und 1 subtrahieren
                STA softroll        ;neuen Scroll-Offs. Abl.
                BPL rollon          ;Wenn >0, dann weiter
```

Wie Sie sehen, wird hier zunächst der IRQ-Zähler für den nächsten Rasterdurchlauf auf 0 zurückgesetzt. Anschließend subtrahieren wir den Wert 1 vom Scroll-Offset, wodurch die Sprites im nächsten Durchlauf eine Y-Position höher dargestellt werden. Durch Ändern des

hier subtrahierten Wertes in 2 oder 3 können Sie übrigens auch die Scrollgeschwindigkeit erhöhen. Gab es bei der Subtraktion keinen Unterlauf, so wird zum Label "ROLLON" (s.u.) verzweigt. Im anderen Fall wurde durch den Scroll soeben eine ganze Spritezeile aus dem Bildschirm gescrollt, weswegen wir die Zeile unten, mit einem neuen Text belegt, wieder einfügen müssen. Zusätzlich müssen die Spritepointer in anderer Reihenfolge ausgelesen werden, was wir durch das Hochzählen von "SHOWLINE" bewirken.
Diese Aufgaben werden in folgendem Programmteil ausgeführt:

```
inc showline        ;1. Spr-Zeilen- Ind. erh.
SEC                 ;C-Bit f. Subtr. Setzen
LDA showline        ;Showline holen
SBC #$0D            ;und 13 subtr.
BMI noloop          ;Bei Unterlauf weiter
STA showline        ;Sonst Wert abl.
```

Da unsere Pointertabelle zwar doppelt, aber nicht ewig lang ist, muß sie natürlich alle 13 Spritezeilen wieder zurückgesetzt werden, was durch den SBC-Befehl geschieht. Erzeugte die Subtraktion ein negatives Ergebnis, so sind wir noch in einer Zeile kleiner als 13, und der erhaltene Wert wird ignoriert. Im andern Fall haben wir soeben die Mitte der Pointerliste erreicht, ab der ja die selben Werte stehen wie am Anfang, und wir können "SHOWLINE" wieder auf den erhaltenen Wert (immer 0) zurücksetzen.
Den nun folgenden Routinenteil, der ab dem Label "NOLOOP" beginnt, möchte ich Ihnen nur der Vollständigkeit halber hier auflisten. Er prüft, ob die Laufschrift, die an dem Label "ESTEXT" abgelegt ist, schon zu Ende gescrollt wurde.
Wenn ja, so wird in diesem Fall der Zeiger "TPOINT", der auf das erste Zeichen der als nächstes darzustellenden Spritezeile zeigt, wieder mit der Startadresse des Textes ("ESTEXT") initialisiert:

```
NOLOOP      LDA tpoint
            CMP #<estext+($34*$18)
            BNE continue
            LDA tpoint+1
            CMP #<estext+($34*$18)
            BNE continue
            LDA #$00
            STA showline
            LDA #<estext
            STA tpoint+0
            LDA #>estext
            STA tpoint+1
```

Es folgt nun der Programmteil mit dem Label "CONTINUE", der von der obigen Routine angesprungen wird. Hier kümmern wir uns wieder um den Scrolloffset, dessen Wert ja noch negativ ist, da wir im ersten Teil der MOVESPR-Routine durch die Subtraktion einen Unterlauf des Zählers erzeugt hatten. Damit Sie hier nun auch Werte größer 1 einsetzen können, womit der Scroll schneller durchläuft, wird "SOFTROLL" nicht wieder mit $17 vorinitialisiert, sondern wir addieren auf das Ergebnis der Subtraktion den Offset, der zwischen zwei Rasterinterrupts liegt, $18(= dez.24), auf. Der resultierende Wert wird dann wieder in Softroll abgelegt. Auf diese Weise werden also auch Scrollwerte größer 1 berücksichtigt. War das Ergebnis der Subtraktion z. B.-2, so wird "SOFTROLL" auf 22 zurückgesetzt, womit der Überlauf abgefangen wird, und der Scrolleffekt flüssig weiterläuft:

```
CONTINUE    CLC             ;C-Bit f.Add. löschen
            LDA softroll    ;SOFTROLL laden
```

```
ADC #$18        ;24 addieren
STA softroll    ;und wieder ablegen
LDA #$20        ;Op-Code für "JSR"
STA mt          ;in MT eintragen
```

Besonders trickreich ist die LDA-STA-Folge am Ende dieses Programmteils. Wir tragen hier den Wert $20, der dem Assembler-Opcode des "JSR"- Befehls entspricht, in das Label "MT" ein. Letzteres befindet sich innerhalb der MOVIEIRQ-Routine, und zwar vor dem Befehl "JSR MAKETEXT". Die MAKETEXT-Routine baut eine Spritezeile auf, indem Sie die Zeichendaten dieser Zeile von dem Zeichensatz bei $0800 in die zu benutzenden Sprites einkopiert. Da dies nicht direkt zu unserem Kursthema gehört, möchte ich auch nicht weiter auf diese Routine eingehen. Wichtig zu wissen ist nur, daß die MOVESPR-Routine, nachdem sie erkannt hat, daß eine neue Spritezeile aufgebaut werden muß, die MOVIEIRQ-Routine derart modifiziert, daß im nächsten IRQ die MAKETEXT-Routine angesprungen wird. Innerhalb selbiger existiert dann eine weitere Befehlsfolge, die den Wert $2C in das Label "MT" schreibt. Selbiger Wert ist der Opcode für den Assemlerbefehl "BIT". In allen folgenden IRQs arbeitet der Prozessor an diesem Label also immer den Befehl "BIT MAKETEXT" ab, der eigentlich keine Funktion beinhaltet, sondern lediglich verhindern soll, daß die Maketext-Routine angesprungen wird. Erst, wenn MOVESPR erkannt hat, daß eine neue Spritezeile aufgebaut werden muß, ändert sie den Befehl wieder in "JSR MAKETEXT" um, so daß die Zeile im nächsten IRQ wieder automatisch neu berechnet wird. Dieser, zugegebenermaßen etwas umständliche, Weg des Routinenaufrufs wurde gewählt, da MAKETEXT recht lange (insgesamt etwa einen Rasterdurchlauf) braucht, um die Spritezeile aufzubauen. Damit Sie in dieser Zeit die Raster-IRQs nicht blockiert, muß sie auf diesem Weg benutzt werden. Während ihres Ablaufs erlaubt sie auch weitere IRQs, so daß Sie während der Darstellung des nächsten Rasterdurchlaufs, immer zwischen den Spritepositionierungen durch den IRQ, ablaufen kann.
Kommen wir nun zum letzten Teil der MOVESPR-Routine, dem Label "ROLLON" . Selbiges wird ja ganz am Anfang der Routine angesprungen, wenn kein Unterlauf von "SOFTROLL" stattfand, und somit ohne jegliche Änderung weiter gescrollt wird.
Sie setzt den Spritepointerindex "ISLINE" zurück auf den Wert in "SHOWLINE" und bereitet den ersten Raster-IRQ vor, der ja immer in der Rasterzeile "SOFT-ROLL" aufzutreten hat.

```
ROLLON  LDA showline    ;ISLINE mit Inhalt von
        STA isline      ;SHOWLINE initialisieren
        LDA softroll    ;Scroll-Offset holen
        STA $D012       ;und als nächsten IRQ-
        LDA $D011       ;Auslöser setzen, dabei
        AND #$7F        ;High-Bit löschen und
        ORA #$08        ;gleichz. 24-Zeilen-
        STA $D011       ;Darst. einschalten
        RTS
```

Beim Festlegen des Wertes "SOFTROLL" als nächste IRQ-Rasterzeile, muß die Routine auch das High-Bit, dieser Rasterposition, in $D011 löschen. Gleichzeitig schaltet sie die 25-Zeilen-Darstellung durch Setzen des 3. Bits dieses Registers wieder ein. Dies ist eigentlich eine Aufgabe, die für das Beispielprogramm "MOVIE.1" irrelevant ist. Wohl aber für "MOVIE.2", in dem wir den Movie- Scroller über einen Bildschirm mit abgeschaltetem oberen und unteren Rand laufen lassen. Wie Sie wissen, muß dazu in Rasterzeile $FA die Darstellung von 25 auf 24 Zeichen-Zeilen heruntergeschaltet werden, und danach, vor nochmaligem Erreichen von $FA, wieder auf 25 Zeilen zurück, was hiermit durchgeführt wird. Da MOVESPR immer im 13. Interrupt aufgerufen wird, und dieser immer nur

innerhalb der Rasterzeilen $10C-$126 auftreten kann, befinden wir uns tatsächlich schon unterhalb der Rasterzeile $FA, womit die Änderung zum korrekten Zeitpunkt durchgeführt wird.

3. Gleichzeitiges Öffnen des Bildschirmes

Wie schon angesprochen, öffnet das Programmbeispiel "MOVIE.2" zusätzlich noch den oberen und unteren Bildschirmrand, damit die Sprites in voller Bildhöhe über den Bildschirm laufen. Vom Prinzip her ist dies ein recht einfaches Unterfangen, das wir auch schon ausgiebig in diesem Kurs besprochen und angewandt haben. Durch unsere Scrollroutine stellt sich uns jedoch ein kleines Problem in den Weg: da unsere Raster-IRQs durch den Scrolleffekt immer in verschiedenen Rasterzeilen aufzutreten haben, und wir nicht immer genau sagen können, ob nun der 11. oder 12. Rasterinterrupt gerade ausgelöst wurde, bevor der Rasterstrahl die Position $FA erreicht hat, wird es schwierig die Scroll-Routine so auszutimen, daß sie genau an dieser Position die Bildschirmdarstellung ändert. Würde man versuchen durch Verzögerungsschleifen die richtige Position abzustimmen, so wäre das mit einem erheblichen Programmieraufwand verbunden. Deshalb greifen wir zu einem kleinen Trick: Die INIT-Routine von "MOVIE.2" wurde um eine kleine Änderung erweitert. Zunächst lassen wir hier den Prozessor, durch ständiges Auslesen und Vergleichen der aktuellen Rasterposition, auf Rasterzeile $FA warten. Ist diese erreicht, so initialisieren wir Timer A von CIA-2 mit dem Wert $4CC7 und starten ihn. Da der Rasterstrahl immer exakt so viele Taktzyklen braucht, um einmal über den ganzen Bildschirm zu laufen, erzeugt diese CIA immer genau in Rasterzeile $FA, in der sie gestartet wurde, einen NMI. Weil dieser Vorrang vor dem IRQ hat, wird er selbst dann ausgelöst, wenn gerade ein Raster-IRQ auftritt oder in Bearbeitung ist. Die NMI-Routine nimmt nun die erforderliche Änderung von $D011 vor, um den Rand abzuschalten (Bit 3 löschen) und kehrt anschließend sofort wieder zurück, ggf. sogar in einen vom NMI unterbrochenen Raster-IRQ, der dann ganz normal zu Ende bearbeitet wird. Das Zurücksetzen von $D011 auf die alte 25-Zeilen-Darstellung, wird dann wieder von der MOVESPR-Routine durchgeführt, wie wir oben ja schon gesehen hatten.
Um die NMIs zu erzeugen springt die INIT-Routine von "MOVIE.2" auf eine Unterroutine zum Initialisieren des Timer- NMIs. Wie das funktioniert hatten wir schon ganz zu Anfang des Interrupt-Kurses besprochen. Hier die Routine, um Ihnen den Vorgang wieder ins Gedächtnis zu rufen. Wie auch schon für den IRQ springen wir diesmal nicht über den Soft-NMI-Vektor bei $0318/$0319, sondern über den Hardvektor bei $FFFA/$FFFB in den NMI ein:

```
NMIINIT   LDA #<nmi       ;Startadresse NMI-Routine
          STA $FFFA       ;in die Hardvektoren bei
          LDA #>nmi       ;$FFFA/$FFFB eintragen
          STA $FFFB
          LDA #$C7        ;Timer A mit dem Zählwert
          STA $DD04       ;$4CC7 initialiseren
          LDA #$4C
          STA $DD05
          LDA #$FA        ;Rasterzeile $FA in Akku
WAIT      CMP $D012       ;mit aktueller Rasterzeile vergleichen
          BNE wait        ;ungleich also weiter
          LDA #$11        ;Gleich, also Timer A
          STA $DD0E       ;starten
          LDA #$81        ;Timer-A-NMIs in ICR
          STA $DD0D       ;erlauben
          RTS
```

Das war eigentlich schon alles. Von nun an löst Timer A von CIA-B alle $4CC7 Taktzyklen

einen NMI aus. Da der Rasterstrahl immer genau diese Anzahl Zyklen benötigt, um ein ganzes Mal über den Bildschirm zu laufen, tritt der NMI also immer in Rasterzeile $FA ein, auf die wir vor Starten des Timers gewartet haben. Die NMI-Routine selbst ist recht kurz und sieht folgendermaßen aus:

```
NMI     PHA             ;Akku retten
        LDA $D011       ;$D011 holen
HILO    ORA #$00        ;7.Bit Rasterpos. setzen
        AND #$F7        ;3.Bit löschen
        STA $D011       ;$D011 zurückschreiben
        BIT $DD0D       ;NMIs wieder erlauben
        PLA             ;Akku zurückholen
        RTI             ;und Ende
```

Da die NMI-Routine lediglich den Akku benutzt, brauchen wir auch ausschließlich nur diesen auf dem Stapel zu retten. Danach wird der Inhalt von $D011 gelesen. Der nun folgende ORA-Befehl hat die Aufgabe eine ggf. gesetztes High-Bit der Rasterstrahlposition des nächsten Raster-IRQs zu setzen, damit wir durch unsere Manipulation von $D011 nicht versehentlich die, schon gesetzte, nächste Raster-IRQ-Position verändern. Hierzu wurde die MOVIEIRQ-Routine von "MOVIE.2" derart abgeändert, daß Sie die High-Position für den nächsten Raster-IRQ nicht nur in $D011 schreibt, sondern auch im Label " HILO+1" ablegt, so daß das Argument des ORA-Befehls zwischen $00 und $80 variiert und immer den richtigen ODER-Wert enthält. Anschließend folgt nun ein AND-Befehl zum Löschen des 3. Bits, womit wir mit dem Ablegen des Wertes die 24-Zeilen-Darstellung einschalten. Durch den BIT-Befehl führen wir einen Lesezugriff auf das ICR- Register von CIA-2 aus, womit wir selbiges Löschen, und dadurch das Auftreten und Melden des nächsten NMIs ermöglichen. Hiernach wird dann nur noch der gerettete Akkuinhalt zurückgeholt, bevor wir den NMI mittels "RTI" beenden. Da die MOVESPR-Routine nun automatisch zu einer späteren Position Bit 3 in $D011 wieder setzt, funktioniert der NMI-Border-Trick also auch wieder im folgenden Rasterdurchlauf! Mit Hilfe des hier benutzen NMI-Tricks können Sie quasi zwei Raster-Interrupts gleichzeitig ablaufen lassen, was auch für andere Rasterroutinen sehr hilfreich sein kann.

(ih/ub)

Teil 11 – Magic Disk 09/94

Herzlich Willkommen zum elften Teil unseres IRQ-Kurses. Wie schon in den letzten Kursteilen, werden wir uns auch diesen Monat mit der Spriteprogrammierung der besonderen Art beschäftigen. Es soll um einige Tricks gehen, mit denen man den Bildschirm auch über alle Ränder hinaus mit Grafikdaten füllen kann. Diesen Effekt nennt man "ESCOS", der Dreh- und Angelpunkt für die Rastertricks in diesem Kursteil sein wird.

1. Unser Ziel

In früheren Kursteilen hatten wir ja schon einmal besprochen, auf welche Art und Weise oberer und unterer, sowie linker und rechter Bildschirmrand abgeschaltet werden. Wir hatten weiterhin gelernt, daß in diesen Bereichen ausschließlich Sprites auftauchen können, die durch die abgeschalteten Ränder sichtbar sind, wo sie sonst von letzteren überdeckt werden. Wir hatten ebenso eine Möglichkeit kennengelernt, beide Ränder, die horizontalen und vertikalen, gleichzeitig abzuschalten, wobei wir auf sehr exaktes Timing achten mussten, da das Abschalten der linken und rechten Bildschirmbegrenzung eine hohe Genauigkeit erforderte. Des weiteren wird Ihnen aus dem letzten Kursteilen bestimmt noch die Sprite-Multiplexer- Routine in Kombination mit einem Moviescroller im Kopf sein,

mit der wir 104 Sprites gleichzeitig auf den Bildschirm brachten. Wie Sie sich vielleicht erinnern, waren wir dabei an gewisse Grenzen gebunden. So war z. B. zwischen zwei Spritezeilen immer ein Abstand von ca.2 Rasterzeilen erforderlich, die wir benötigten, um die Spritepointer sowie die neuen Y-Positionen der Sprites zu setzen. Des weiteren mussten wir ein komplizierte Timingroutine benutzen, die zwischen Charakterzeilen (jede achte Rasterzeile, in der der VIC den Prozessor für eine Dauer von 42 Taktzyklen anhält) und normalen Rasterzeilen zu unterscheiden hatte. In dieser Folge unseres Kurses wollen wir nun all diese Komponenten miteinander verbinden und eine Möglichkeit kennenlernen, Timingprobleme durch Charakterzeilenberücksichtigung, zu umgehen. Das Endergebnis wird ein flächendeckend (!) mit Sprites belegter Bildschirm sein, wobei weder Leerräume zwischen den Sprites, noch im gesamten Bildschirmrahmen zu sehen sein werden!
Wir werden also über eine Grafik verfügen, die, ähnlich einem Fernsehbild, die gesamte Bildröhrenfläche ausfüllt!

2. Erstes Problem: Das Timing

Kommen wir gleich zum Kern dieses Kursteils, dem Timing-Problem, das sich uns entgegenstellt. Möchten wir nämlich alle Bildschirmränder abschalten und gleichzeitig auch noch die Sprites multiplexen, so können wir programmtechnisch in " Teufels Küche" gelangen. Wir müssten berücksichtigen, daß im sichtbaren Bildschirmfenster alle acht Rasterzeilen eine Charakterzeile auftritt, gleichzeitig müsste in jeder Rasterzeile der linke und rechte Bildschirmrand abgeschaltet, sowie in jeder 21 . Rasterzeile die Sprites neu positioniert werden. Dabei stellen vor allem die Charakterzeilen ein großes Problem dar. Wir müssten unterscheiden zwischen Rasterstrahlpositionen im oberen und unteren Bildschirmrand und innerhalb des sichtbaren Bildschirmfensters, und zudem noch für letzteren Fall berücksichtigen, wann sich der VIC gerade in einer Charakterzeile befindet und wann in einer normalen Rasterzeile. Dieses Problem stellte sich bei unseren Raster-Effekten nun schon häufiger in den Weg, wobei wir es meist durch einen einfachen Trick umgingen: in der Regel benutzten wir in solchen Fällen eine FLD-Routine, die die Charakterzeile im fraglichen Bildschirmbereich einfach nach unten "wegdrückte", so daß wir in jeder Rasterzeile 63 Taktzyklen zur Verfügung hatten und somit ein einheitliches Timing programmieren konnten.
Wir könnten diesen Effekt hier nun auch anwenden, jedoch gibt es speziell für diese Anwendung einen weiteren, viel einfacheren Trick, die Charakterzeilen zu umgehen: da wir ja den gesamten Bildschirm mit Sprites füllen möchten, können wir davon ausgehen, daß wir keinerlei Hintergrundgrafik, bzw. Textzeichen benötigen. Es gibt nun einen Trick, den wir noch nicht kennengelernt haben, mit dem wir die Charakterzeilen ganz abschalten können, so daß nur noch die Sprites dargestellt werden. Wie fast immer ist Register $D011, ein Dreh- und Angelpunkt der VIC-Trickeffekt-Kiste, für diesen Trick verantwortlich. Mit Bit4 dieses Registers können wir nämlich den gesamten Bildschirm abschalten, was einer Deaktivierung des VICs gleichkommt. Er wird durch das Löschen dieses Bits veranlasst, auf dem gesamten Bildschirm nur noch die Rahmenfarbe darzustellen, und keine Charakterzeilen mehr zu lesen. Die Sprites bleiben jedoch weiterhin aktiv, obwohl sie jetzt unsichbar sind, da sie jetzt auf dem gesamten Bildschirm mit dem Rahmen überdeckt werden. Man könnte das Setzen und Löschen des 4. Bits von Register $D011 quasi mit dem Öffnen und Schließen eines Vorhangs vor einem Fenster vergleichen: Eine Fliege (oder unser Sprite), die sich auf der Fensterscheibe befindet, ist bei geschlossenem Vorhang nicht sichtbar. Ist letzterer jedoch geöffnet, so sieht man die Fliege, solange sie sich im sichtbaren Bereich des Fenster bewegt, und nicht unter den seitlich aufgerafften Vorhängen verschwindet.
Nun, selbst wenn die Sprites noch aktiv sind, so nutzen Sie uns herzlich wenig wenn sie unsichtbar sind. Deshalb gilt es mal wieder, den VIC auszutricksen, um das gewünschte Ergebnis zu erhalten.

Dies gestaltet sich in diesem Fall recht einfach: Zunächst einmal schalten wir an einer Rasterposition, an der der VIC normalerweise den oberen oder unteren Bildschirmrand zeichnet, den gesamten Bildschirm durch Löschen von Bit 4 aus Register $D011, ab. Erreicht der Rasterstrahl nun Rasterzeile $30, an der eigentlich das sichtbare Bildschirmfenster beginnt, so prüft der VIC, ob der Bildschirm nun ein- oder ausgeschaltet ist. In letzterem Fall deaktiviert er seine Zeichenaufbau-Schaltkreise bis zum nächsten Erreichen dieser Rasterposition, womit er keine einzige Charakterzeile mehr liest. Anstelle dessen zeigt er in den folgenden Rasterzeilen nur noch die Farbe des Bildschirmrahmens an. Wenn wir diesen jedoch mit Hilfe einer Border-Routine bei $FA abschalten, so - oh Wunder - zeigt uns der VIC den Bildschirmhintergrund, auf dem sich auch die Sprites herumtollen dürfen! Wie bei jedem Effekt, bei dem der VIC etwas tut, was er sonst nicht tun kann, erscheint hier dann wieder der Inhalt der letzten Adresse des VIC-Speichers (normalerweise $3FFF) in Schwarz auf Hintergrundfarbe.
Durch Schreiben des Wertes 0 in diese Speicherzelle können wir natürlich die Schwarzen Streifen auch abschalten und damit nur die Hintergrundfarbe anzeigen lassen.
Um diese Vorgehensweise nun besser zu erläutern haben wir natürlich wieder ein Programmbeispiel auf Lager, das ich Ihnen nun auflisten möchte. Sie finden es auf dieser MD auch als fertig ausführbares File mit dem Namen "SPRITES-ONLY", daß Sie wie immer mit ",8,1" laden und durch ein "SYS4096" starten müssen.
Kommen wir also zur Initialisierungsroutine des Beispiels, die bei Adresse$1000 beginnt:

```
INIT    SEI             ;IRQs sperren
        LDA #$7F        ;Alle CIA-IRQs abschalten
        STA $DC0D       ;(CIA1)
        STA $DD0D       ;(CIA2)
        BIT $DC0D       ;CIA1-ICR löschen
        BIT $DD0D       ;CIA2-ICR löschen
        LDA #$01        ;VIC-Hintergrundstriche
        STA $3FFF       ;auf 1 setzen
        JSR sprinit     ;Sprites initialisieren
        JSR irqinit     ;IRQ initialisieren
        LDA #$35        ;ROMs abschalte
        STA $01
        CLI             ;IRQs erlauben
spc     LDA #$7F        ;Auf SPACE-Taste
        STA $DC00       ; warten...
        LDA $DC01
        CMP #$EF
        BNE spc
        SEI             ;IRQs sperren
        LDA #$37        ;ROM wieder einschalten
        STA $01
        JMP $FCE2       ;und RESET auslösen
```

Alles in allem für uns keine besondere Initialisierung. Wichtig sind noch die Routinen "SPRINIT" und "IRQINIT". In ersterer initialisieren wir lediglich die Sprite-Positionen, sowie die Sprite-Pointer und schalten alle acht Sprites ein. Letztere Routine ist für das Korrekte initialisieren unseres IRQs zuständig und sieht folgendermaßen aus:

```
IRQINIT LDA #< bordirq  ;IRQ-Vektor bei
        STA $FFFE       ;$FFFE/$FFFF
        LDA #> bordirq  ;auf "BORDIRQ" verbiegen
        STA $FFFF
        LDA #$1B        ;Wert für $D011 mit gel.
        STA $D011       ;High-Bit f. Rasterpos.
        LDA #$FA        ;Ersten Raster-IRQ bei
```

```
                STA $D012       ;Zeile $FA auslösen
                LDA #$81        ;VIC-Raster-IRQs
                STA $D01A       ;erlauben
                DEC $D019       ;VIC-ICR ggf. löschen
                RTS
```

Wie Sie sehen, aktivieren wir hier einen Raster-IRQ für Rasterzeile $FA, der bei Auftreten die Routine "BORDIRQ" anspringt, wo sich eine ganz gewöhnliche Routine zum Abschalten des oberen und unteren Bildschirmrandes befindet. Hier das Listing dieser Routine:

```
BORDIRQ         PHA             ;Prozessorregister. retten
                TXA
                PHA
                TYA
                PHA
                LDA $D011       ;24-Zeilen-Darstellung
                AND #$77        ;einschalten
                STA $D011
                LDA #$28        ;nächster IRQ bei Zeile $28
                STA $D012
                DEC $D019       ;VIC-ICR löschen
                LDA #<soff      ;Routine für nächsten IRQ
                STA $FFFE       ;"SOFF" in IRQ-Vektor
                LDA #>soff      ;eintragen
                STA $FFFF
                PLA             ;Prozessorregs. wieder vom
                TAY             ;Stapel holen und IRQ
                PLA             ;beenden
                TAX
                PLA
                RTI
```

Wir schalten hier also an der üblichen Position auf 24-Zeilen-Darstellung herunter, damit der VIC vergisst, den oberen und unteren Bildschirmrand zu zeichnen. Gleichzeitig wird ein neuer IRQ initialisiert, der bei Erreichen von Rasterzeile $28 (acht Rasterzeilen vor Beginn des sichtbaren Bildschirmfensters) die Routine "SOFF" anspringen soll. Diese Routine übernimmt nun die Aufgabe, die Darstellung der Charakterzeilen zu verhindern:

```
soff            PHA             ;Prozessorregister retten
                TXA
                PHA
                TYA
                PHA
                LDA $D011       ;Bild ausschalten (durch
                AnD #$6F        ;ausmaskieren von Bit4)
                STA $D011
                LDA #$32        ;nächster IRQ bei Raster-
                STA $D012       ;zeile $32
                DEC $D019       ;VIC-ICR löschen
                LDA #<son       ;Nächster IRQ soll auf
                STA $FFFE       ; Routine "SON" springen
                LDA #>son
                STA $FFFF
                PLA             ;Prozessorregs. wieder vom
                TAY             ;Stapel holen und IRQ
                PLA             ;beenden
                TAX
```

```
        PLA
        RTI
```

Wie Sie sehen eine recht einfache Aufgabe: durch eine AND-Verknüpfung des $D011-Inhalts mit dem Wert $6F wird einfach das 4. Bit dieses Registers gelöscht. Gleichzeitig löschen wir dabei Bit 7, das ja das High-Bit der Rasterstrahlposition angibt, womit wir also auch dieses Bit für den folgenden Raster-IRQ bei Zeile $32 vorbereitet hätten. Die Routine, die hier abgearbeitet werden soll, heisst "SON" und ist für das Wiedereinschalten des Bildschirms verantwortlich. Da sich Rasterzeile $32 im sichtbaren Fenster befindet wäre somit der VIC überlistet und soweit gebracht, daß er keine Charakterzeilen mehr liest, geschweige denn darstellt.
Gleichzeitig schaltet diese Routine wieder auf die 25-Zeilen-Darstellung zurück (Bit 3 von $D011 muß gesetzt werden), damit das Abschalten des Borders auch im nächsten Rasterstrahldurchlauf funktioniert:

```
SON     PHA             ;Prozessorregister retten
        TXA
        PHA
        TYA
        PHA
        LDA $D011       ;Bild und 25- Zeilen
        ORA #$18        ;Darstellung einschalten
        STA $D011
        LDA #$FA        ;nächster IRQ wieder bei
        STA $D012       ;Zeile $FA
        DEC $D019       ;VIC-ICR löschen
        LDA #<bordirq   ;Wieder "BORDIRQ" in
        STA $FFFE       ;IRQ-Vektor eintragen
        LDA #>bordirq
        STA $FFFF
        PLA             ;Prozessorregs. wieder vom
        TAY             ;Stapel holen und IRQ
        PLA             ;beenden
        TAX
        PLA
        RTI
```

Nachdem Register $D011 auf den gewünschten Wert zurückgesetzt wurde, wird der IRQ wieder für die Routine "BORDIRQ" bei Rasterzeile $FA vorbereitet, womit sich der Kreis schließt und unser Programmbeispiel komplett ist.

3."ESCOS"- einen Schritt weiter

Nachdem wir nun unser Timing-Problem gelöst, und die störenden Charakter-Zeilen aus dem Weg geräumt haben, möchten wir wieder zu unserer eigentlichen Aufgabenstellung zurückkehren: dem bildschirmfüllenden Darstellen von Sprites.
Hierzu müssen wir, nachdem oberer und unterer Bildschirmrand, sowie die Charakterzeilen abgeschaltet wurden, zusätzlich auch noch die Bildschirmränder links und rechts deaktivieren. Das Funktionsprinzip einer solchen Sideborderroutine sollte Ihnen noch aus einem der ersten Kursteile im Kopf sein: durch rechtzeitiges Umstellen von 40- auf 38-Spaltendarstellung und zurück tricksen wir den VIC nach dem selben Prinzip aus, wie wir es bei den horizontalen Bildschirmrändern tun. Dadurch aber, daß sich der Rasterstrahl horizontal recht schnell bewegt, kommt es dabei auf ein höchst exaktes Timing an. Einen Taktzyklus zu früh oder zu spät funktioniert die Routine schon nicht mehr. Wenn man das Ganze nun zusätzlich noch mit dem Öffnen des oberen und unteren Randes, sowie dem Abschalten der Charakter-Zeilen und einem Sprite-Multiplexer

kombinieren muß, so könnte man meinen, daß dies eine recht programmieraufwendige Sache werden kann. Doch keine Panik, die Umsetzung ist einfacher als Sie glauben.
Am Besten sehen Sie sich erst einmal das Demoprogramm " ESCOS1" an. Es wird wie üblich geladen und mit SYS4096 gestartet, und zeigt dann einen vollends mit Sprites gefüllten Bildschirm - und zwar über alle Ränder hinaus! Um so etwas nun sebst zu programmieren, werden wir zunächst auf die organisatirischen Probleme und deren Lösung eingehen:
Als Erstes müssen Sie davon ausgehen, daß wir keine IRQ-Routine im eigentlichen Sinne programmieren werden. Aufgrund des exakten Timings, das zum Abschalten des linken und rechten Randes notwendig ist, muß der Prozessor nahezu während des gesamten Rasterdurchlaufs damit beschäftigt sein, die richtige Rasterposition abzuwarten, um zwischen der 38- und 40-Zeilen-Darstellung hin- und herzuschalten. Dadurch wird lediglich ein einziger IRQ pro Rasterdurchlauf aufgerufen, nämlich direkt zu Anfang desselben, in Rasterzeile 0. Dieser IRQ muß nun, auf die uns schon bekannte Art und Weise,"geglättet" werden, so daß kein einziger Taktzyklus Unterschied zum vorherigen Rasterdurchlauf besteht.
Ab dann (durch das Glätten, das 2 Rasterzeilen dauert also ab Zeile 2) beginnt der Prozessor damit in jeder einzelnen Rasterzeile den Rand abzuschalten. Dies geschieht dadurch, daß wir nach jedem Umschalten der Spaltendarstellung genau am Übergangspunkt zwischen sichtbarem Bildschirmfenster und - rahmen, exakt 63 Taktzyklen verzögern, bevor dieser Vorgang wiederholt wird. So zumindest sähe es aus, wenn wir keine Sprites darzustellen hätten. Da wir dies jedoch tun, müssen wir weiterhin berücksichtigen, daß der VIC zum Darstellen der Sprites den Prozessor ebenso anhalten muß, wie beim Lesen der Charakterzeilen, um sich die Spritedaten aus dem Speicher zu holen. Hierbei braucht er 3 Taktzyklen für das erste Sprite, und dann jeweils 2 Zyklen für alle weiteren, eingeschalteten Sprites. Da wir alle 8 Sprites benutzen, müssen wir also noch den Betrag 3+7*2=17 von den 63 Zyklen abziehen und erhalten somit exakt 46 Taktzyklen, die der Prozessor pro Rasterzeile " verbrauchen" muß.

Gleichzeitig müssen wir berücksichtigen, daß alle 21 Rasterzeilen die Y-Position der Sprites um diesen Betrag hochgezählt werden muß, damit sie auch untereinander auf dem Bildschirm erscheinen. Dies ist glücklicherweise eine nicht allzu schwere Aufgabe, da der VIC uns diesbezüglich ein wenig entgegenkommt. Ändern wir nämlich die X-Position eines Sprites innerhalb einer Rasterzeile, so hat dies eine sofortige eine Wirkung auf die Spriteposition: das Sprite erscheint ab dieser Rasterzeile an der neuen X-Position. Mit der Y-Position verhält es sich anders:
wird sie noch während das Sprite gezeichnet wird geändert, so hat das keine direkte Auswirkung auf die restlichen Spritezeilen. Der VIC zeichnet das Sprite stur zu Ende, bevor er die Y-Position noch einmal prüft. Das gibt uns die Möglichkeit, die Y-Position schon im Vorraus zu ändern, nämlich irgendwann innerhalb der 21 Rasterzeilen, die das Sprite hoch ist. Setzen wir die neue Y-Position nun also genau auf die Rasterzeile nach der letzten Spritezeile, so kümmert sich der VIC darum erst einmal gar nicht. Er zeichnet zunächst sein aktuelles Sprite zu Ende, und wirft dann erst einen Blick in das Y-Positions-Register des Sprites. Da er dort dann eine Position vorfindet, die er noch nicht überlaufen hat, glaubt er, das Sprite wäre noch nicht gezeichnet worden, woraufhin er unverzüglich mit dem Zeichnen wieder anfängt - und schon wäre dasselbe Sprite zweimal untereinander auf dem Bildschirm zu sehen! Dadurch können wir uns in der Rasterroutine mit der Neupositionierung Zeit lassen, und selbige über zwei Rasterzeilen verteilt durchführen (innerhalb einer Zeile wäre auch gar nicht genug Zeit dafür).
Um die Bildschirmränder unsichtbar zu machen muß sich ein Teil unserer IRQ-Routine um das Abschalten des oberen und unteren Bildschirmrandes, sowie der Charakterzeilen kümmern. Wie Sie im Programmbeispiel zuvor gesehen haben, ist das eigentlich eine

recht einfache Aufgabe. Da wir zum Abschalten der Ränder links und rechts jedoch ein hypergenaues Timing benötigen, wäre es recht aufwendig für die Rasterzeilen $FA,$28 und $32 eigene IRQ-Routinen zu schreiben, ohne dabei das Timing für die Sideborderabschaltung damit durcheinander zu bringen. Aus diesem Grund haben wir uns einen Trick ausgedacht, mit dem wir beide Aufgaben quasi "in einem Aufwasch" bewältigen: Wie Sie zuvor vielleicht schon bemerkt haben, hatten alle IRQs unserer Beispielroutine eins gemeinsam:
Sie erzielten den gewünschten Effekt durch irgendeine Manipulation des Registers $D011 . Warum sollten wir also nicht aus zwei eins machen, und generell in jeder Rasterzeile einen Wert in dieses Register schreiben? Das hilft uns zum Einen das Verzögern des Rasterstrahls bis zum Ende der Rasterzeile, und hat zudem den angenehmen "Nebeneffekt" die horizontalen Bildschirmränder, sowie die Charakterzeilendarstellung quasi "automatisch" abzuschalten.
Nach all dieser trockenen Theorie möchte ich Ihnen den Source-Code zu unserer ESCOS-Routine nun nicht mehr länger vorenthalten. Die Init-Routine werden wir uns diesmal sparen, da sie nahezu identisch mit der des letzten Programmbeispiels ist. Wichtig für uns ist, daß Sie den VIC darauf vorbereitet, einen IRQ in Rasterzeile 0 auszulösen, bei dessen Auftreten der Prozessor in die Routine "SIDEBORD" zu springen hat. Selbige Routine befindet sich an Adresse $1200 und sieht folgendermaßen aus:

```
SIDEBORD    PHA             ;Prozessorregister retten
            TXA
            PHA
            TYA
            PHA
            DEC $D019       ;VIC-ICR löschen
            INC $D012       ;nächste Rasterz. neuer
            LDA #<irq2      ;IRQ-Auslöser, mit Sprung
            STA $FFFE       ;auf "IRQ2"
            CLI             ;IRQs erlauben
ch          NOP             ;13 NOPs während der der
            …               ;IRQ irgendwann auftritt
            NOP
            JMP ch
IRQ2        LDA #$FF        ;Alle Sprites, sowie
            STA $D015       ;X-Expansion
            STA $D01D       ;einschalten
            CLCc
            LDA $00FB       ;"SOFTROLL" lesen
            ADC #$02        ;Die Y-Position
            STA $D001       ;der Sprites
            STA $D003       ;befindet sich
            STA $D005       ;2 Rasterzeilen
            STA $D007       ;nach RasterIRQ
            STA $D009       ;und muß demnach
            STA $D00B       ;gesetzt werden.
            STA $D00D
            STA $D00F
            LDA $D012       ;den letzen Zyklus
            CMP $D012       ;korrigieren
            BNE onecycle
ONECYCLE    PLA             ;Vom 2. IRQ erzeugte
            PLA             ;Rücksprungadresse und CPU-
            PLA             ;Status v.Stapel entf.
            LDA #<sidebord  ;IRQ-Ptr. wieder auf
            STA $FFFE       ;"SIDEBORD" zurück
            DEC $D019       ;VIC-ICR löschen
```

```
LDA $FB          ;Index für $D011-
LSR              ;Tabelle vorbereiten
TAY
```

Soweit also keine uns besonders unbekannte Routine. Der hier aufgezeigte Auszug dient lediglich dem Glätten des IRQs und sollte Ihnen, wenn auch leicht modifiziert, schon aus anderen Beispielen bekannt sein. Einzig zu erwähnender Punkt wäre die Adresse $FB. In dieser Zeropageadresse steht die Nummer der Rasterzeile, in der der IRQ aufgerufen werden soll. Obwohl das bei uns zwar immer der Wert 0 ist (womit also immer 0 in diesem Register steht), hat dies einen entscheidenden Vorteil: durch einfaches Hochzählen dieses Registers um 1, nach jedem Rasterdurchlauf, wird der Raster-IRQ jeweils eine Zeile später erst auftreten, womit wir einen ganz simplen Scrolleffekt erzielen. Zu sehen ist dies auch im Beispiel "ESCOS2", das absolut identisch zu "ESCOS1" ist, jedoch mit der oben genannten Änderung.
Gleichzeitig hat die Adresse $FB noch eine zweite Aufgabe: sie wird zusätzlich als Index auf eine Liste "mißbraucht", in der die Werte stehen, die in $D011 eingetragen werden müssen, um oberen und unteren Bildschirmrand, sowie Charakterzeilen abzuschalten. Da im Prinzip nur drei verschiedene Werte in dieser Tabelle stehen, und zudem die Änderungen der Werte nicht hundertprozentig genau in einer speziellen Rasterzeile auftreten müssen, sondern ruhig auch einmal eine Rasterzeile früher oder später, haben wir die Tabelle nur halb solang gemacht und tragen nur jede zweite Rasterzeile einen Wert von ihr in $D011 ein. Dies hat den zusätzlichen Vorteil, daß sie nicht länger als 256 Byte wird und somit keine High-Byte- Adressen berücksichtigt werden müssen. Aus diesem Grund wird also auch der Index-Wert, der ins Y-Register geladen wird, durch einen "LSR"-Befehl halbiert, damit bei einer Verschiebung automatsich die ersten Tabellenwerte übersprungen und somit auf den " richtigen" ersten Tabelleneintrag zugegriffen wird. Wichtig ist nur noch, daß die Tabelle unbedingt innerhalb eines 256- Byte-Blocks steht und an einer Adresse mit Low-Byte- Wert 0 beginnt.
Durch die Y-indizierte Adressierung, mit der wir auf sie zugreifen, könnte es im anderen Fall zu Timing-Problemen kommen.
Greifen wir nämlich mit dem Befehl " LDA $11FF,Y" auf eine Speicherzelle zu, und enthält in diesem Fall das Y-Register einen Wert größer 1, so tritt ein Überlauf bei der Adressierung auf ($11 FF+1=$1200- High-Byte muß hochgezählt werden!). Ein solcher Befehl benötigt dann nicht mehr 4 Taktzyklen, sondern einen Taktzyklus mehr, in dem das High-Byte korrigiert wird! Dadurch würden wir also unser Timing durcheinanderbringen, weswegen die Tabelle ab Adresse $1500 abgelegt wurde und mit Ihren 156 Einträgen keinen solchen Überlauf erzeugt. Die Tabelle selbst enthält nun, jeweils blockweise, die Werte $10,$00 und $18, die an den richtigen Rasterpositionen untergebracht wurden, so daß die Ränder und Charakterzeilen beim Schreiben des entsprechenden Wertes abgeschaltet werden. Sie können ja einmal mit Hilfe eines Speichermonitors einen Blick hineinwerfen.
Wir werden uns nun um den Rest unserer Routine kümmern, in der wir in 294 Rasterzeilen die seitlichen Ränder abschalten und zudem jede 21. Rasterzeile die Sprites neu positionieren, so daß insgesamt 14 Zeilen zu je 8 Sprites auf dem Bildschirm erscheinen. Dies ist die Fortsetzung der SIDEBORD-IRQ- Routine:

```
NOP              ;Diese Befehlsfolge wird
JSR open21       ;insgesamt 14 x aufgerufen
...              ;um je 21 Zeilen zu öffnen
...
NOP              ;Sprite Line 14
JSR open21
LDA #$00         ;Sprites aus
STA $D015
```

```
                LDA #$C8        ;40- Spalten-Modus zurück
                STA $D016       ;setzen
                LDA $FB         ;Zeile aus $FB für nächsten
                STA $D012       ;Raster-IRQ setzen
                PLA             ;Prozessorregs. wieder vom
                TAY             ;Stapel holen und IRQ
                PLA             ;beenden
                TAX
                PLA
                RTI
```

Wie Sie sehen besteht der Kern unserer Routine aus den 14 Mal aufeinander folgenden Befehlen:"NOP" und "JSR OPEN21".
Der NOP-Befehl dient dabei lediglich dem Verzögern. Die Unterroutine "OPEN21" ist nun dafür zuständig, in den folgenden 21 Rasterzeilen (genau die Höhe eines Sprites) den Rand zu öffnen, sowie die neuen Y-Spritepositionen zu setzen. Sie sieht folgendermaßen aus:

```
;line 00
        OPEN21  NOP             ;Verzögern bis rechter
                NOP             ;Rand erreicht
                DEC $D016       ;38 Spalten
                INC $D016       ;40 Spalten
                LDA $1500,Y     ;$D011-Wert aus Tabelle
                STA $D011       ;lesen und eintragen
                INY             ;Tabellen-Index+1
                JSR cycles+5    ;24 Zyklen verzögern
;line 01
                DEC $D016       ;38 Spalten
                INC $D016       ;40 Spalten
                JSR cycles      ;34 Zyklen verzögern
                …               ;dito für 2-15
```

Wie Sie sehen können, verzögern wir zunächst mit zwei NOPs bis zum Ende des sichtbaren Bildschirmfensters, wo wir durch herunter zählen von Register $D016 die 38-Spalten-Darstellung einschalten, und gleich darauf durch Hochzählen desselben Registers wieder auf 40 Zeilen zurückgehen. Damit wäre der Rand geöffnet worden. Als nächstes wird der Wert für $D011 aus der besagten Tabelle ausgelesen und in dieses Register eingetragen, sowie der Index-Zähler im Y-Register für die übernächste Zeile um 1 erhöht. Danach wird die Routine "CYCLES" aufgerufen, jedoch nicht an ihrer eigentlichen Adresse, sondern 5 Bytes weiter. Diese Routine besteht aus insgesamt 11 NOPs, die lediglich die Zeit verzögern sollen, bis die nächste Rand-Abschaltung fällig wird. Da ein NOP 2 Taktzyklen verbraucht, verzögert sie 22 Taktyklen. Hier muß man zusätzlich noch die Zeit hinzurechnen, die für den JSR und RTS-Befehl draufgehen. Beide verbrauchen je 6 Taktzyklen, womit die "CYCLES"- Routine insgesamt 34 Zyklen in Anspruch nimmt. Durch den Offset von 5 Bytes, den wir beim ersten Einsprung in die Routine machen," übergehen" wir einfach die ersten 5 NOPs, womit nur 24 Zyklen verbraucht werden. Nach Rückkehr aus dieser Routine befindet sich der Rasterstrahl nun genau an der Position, an der wir den Rand für die neue Rasterzeile öffnen müssen, was sogleich auch durch die INC-DEC-Befehlsfolge getan wird. Anschließend wird wieder verzögert, wobei wir diesmal die "CYCLES"- Routine komplett durchlaufen, da in dieser Zeile kein neuer Wert in $D011 eingetragen wird, und wir somit 10 Zyklen mehr zu verzögern haben!
Dieser Programm-Auszug wiederholt sich nun insgesamt noch 7 Mal, bis wir in die 16 . Spritezeile gelangt sind. Hier nun beginnen wir mit dem Neupositionieren der Sprites:

```
;line 16
                DEC $D016       ;Altbekannte Methode
                INC $D016       ;um Zeile 16 zu öffnen
                LDA $1500,Y     ;und $D011 neu zu setzen
                STA $D011
                INY
                JSR cycles+5
;line 17
                DEC $D016       ;38 Spalten
                INC $D016       ;40 Spalten
                CLC             ;Neue Y-Pos. für nächste
                LDA $D001       ;Sprite-Reihe= Alte Y-Pos.
                ADC #$15        ;plus 21
                STA $D001       ;Und für Sprites 0-3
                STA $D003       ;eintragen
                STA $D005
                STA $D007
                NOP             ;Nur 10 Zyklen verzögern
                NOP
                NOP
                NOP
                NOP
;line 18
                DEC $D016       ;Altbekannte Methode zum
                INC $D016       ;öffnen der 17. Zeile
                LDA $1500,Y
                STA $D011
                INY
                JSR cycles+5
;line 19
                DEC $D016       ;38 Spalten
                INC $D016       ;40 Spalten
                LDA $D009       ;Sprite-Reihe berechnen
                ADC #$15
                STA $D009       ;und für Sprites 4-7
                STA $D00B       ;setzen
                STA $D00D
                STA $D00F
                NOP             ;Nur 10 Zyklen verzögern
                NOP
                NOP
                NOP
                NOP
;line 20
                DEC $D016       ;38 Spalten
                INC $D016       ;40 Spalten
                LDA $1500,Y     ;Neuen Wert für $D011 aus
                STA $D011       ;Tab. holen u. eintragen
                INY             ;Tab-Index+1
                NOP             ;6 Zyklen verzögern
                NOP
                NOP
                RTS             ;und fertig!
```

Wie Sie sehen, benutzen wir für die geraden Zeilennummern die alte Befehlsfolge zum Öffnen des Randes und Setzen des neuen $ D011- Wertes. In den Spritezeilen 17 und 19 jedoch wird nach dem Öffnen des Randes die Y-Position um 21 erhöht und in je vier Y-Sprite- Register eingetragen, womit also die Y-Positionen der nächsten Spritereihe

festgelegt wären.
In Spritezeile 20 wird nun ein letztes Mal der $ D011-Wert neu gesetzt und nach einer kurzen Verzögerung wieder zum Hauptprogramm zurückgekehrt, von wo die Routine für alle 14 Spritereihen nochmal aufgerufen wird, und womit unser ESCOS-Effekt fertig programmiert ist!

4. Abschließende Hinweise
Wie schon erwähnt, ist der Prozessor zur Anzeige dieser insgesamt 114 Sprites, sowie dem Abschalten der Bildschirmränder, fast den gesamten Rasterdurchlauf lang voll beschäftigt. Viel Rechenzeit bleibt uns nicht mehr übrig, um weitere Effekte zu programmieren. Dennoch sind am Ende des Rasterdurchlaufs noch 16 Rasterzeilen zur freien Verfügung, und man muß noch die Zeit hinzurechnen, in der der Rasterstrahl vom unteren Bildschirmrand wieder zum oberen Bildschirmrand zu wandern hat, in der wir ebenfalls noch mit dem Prozessor arbeiten können. Diese Zeit wurde im Beispiel "ESCOS2" benutzt, um die Spritereihen zusätzlich noch zu scrollen. Jedoch ist auch noch ein Moviescroller problemlos machbar. Dieses Thema werden wir jedoch erst nächsten Monat ansprechen, und uns auch weiterhin mit den Sprites befassen.
Sie werden dabei eine Möglichkeit kennenlernen, wie man diese kleinen Grafikwinzlinge hardwaremässig dehnen und verbiegen kann. Wie immer gibt es dabei einen Trick, den VIC zu überreden das Unmögliche möglich zu machen…

(ih/ ub)

Teil 12 – Magic Disk 10/94

Herzlich Willkommen zu einer neuen Folge unseres IRQ-Kurses. In dieser Ausgabe werden wir uns weiter mit der ESCOS-Routine des letzten Kursteils beschäftigen. Wir werden hierbei die Routine mit einem Movie-Scroller verbinden, um so einen bildschirmübergreifenden Text scrollen zu lassen. Dies gestaltet sich ohne einen speziellen Trick gar nicht mal so einfach…

1. Einleitung
Wie Sie sich vielleicht noch erinnern, so hatten wir in der letzten Folge des IRQ-Kurses zuletzt das Beispielprogramm "ESCOS2" besprochen. Diese Raster-IRQ-Routine öffnete uns den linken und rechten, sowie den oberen und unteren Bildschirmrand und stellte dann alle 21 Rasterzeilen eine neue Reihe mit je acht Sprites auf dem Bildschirm dar. Dadurch hatten wir insgesamt vierzehn Zeilen zu je acht Sprites auf dem Bildschirm, die alle nahtlos aneinandergereiht fast den gesamten Bildschirm überdeckten. Der einzige ungenutzte Bereich war der von Rasterzeile 294 bis 312, der zu klein war um eine weitere Spritereihe darin unterzubringen, aber sowieso schon unterhalb der für normale Monitore darstellbaren Grenze liegt.
Der einzige Unterschied der Routine "ESCOS2" zu "ESCOS1" bestand nun darin, daß erstere Routine lediglich ein Zeilenoffset-Register für den ersten auszulösenden Raster-IRQ herunterzählte, um so einen Scroll-Effekt der Spritereihen zu erzielen."ESCOS2" ist also schon eine Art "Moviescroller". Der Grund, warum sie es nicht wirklich ist, liegt darin, daß in jeder Spritezeile dieselben Sprites auf dem Bildschirm zu sehen waren.
Wie wir aber von den Moviescroll-Routinen wissen, müssen wir für einen Scrolltext das Aussehen einer Spritezeile ja individuell bestimmen können, damit auch wirklich ein Text, und nicht immer dieselbe Zeile, über den Bildschirm läuft.

2. Das Problem - Die Lösung
"Kein Problem" werden Sie nun sagen,"einfach die Textausgaberoutine der Moviescroller-

Routinen vom IRQ-Kurs einbauen, und schon haben wir einen Scrolltext". Diese Feststellung stimmt schon, wie sollte es auch anders gehen, jedoch stellt sich uns dabei noch ein kleines Problem in den Weg: dadurch nämlich, daß wir in den ESCOS-Routinen, keine Zwischenräume mehr zwischen den einzelnen Spritezeilen haben, gestaltet es sich als schwierig, die Sprite-Pointer zeitgenau zu setzen. Setzen wir diese nämlich auf die neu einzuscrollende Spritezeile, noch bevor die letzte Spritezeile abgearbeitet wurde, so erscheinen die alten Sprites schon im Aussehen der Neuen. Umgekehrt können die Spritepointer auch nicht nach Abarbeiten der letzten Spritezeile gesetzt werden da hier ja schon die neuen Sprites erscheinen müssen. Man könnte nun versuchen, ein megagenaues Timing zu programmieren, das innerhalb der ersten Rasterzeile einer neuen Spritezeile exakt vor Darstellen eines neuen Sprites dessen Pointer neu setzt. Dies gestaltet sich jedoch umso umständlicher, wenn wir beachten müssen, daß gleichzeitig auch noch der linke und rechte Rand geöffnet werden soll. Da dieser ebenfalls ein exaktes Timing verlangt, würden wir mit dem Doppeltiming in des Teufels Küche kommen. Aber keine Sorge: glücklicherweise können wir als IRQ-Magier wieder in die Raster-Trickkiste greifen, und eine weitaus einfachere Lösung des Problems anwenden.
Wie Sie z. B. schon von unseren FLI-Routinen wissen, hat man mit dem VIC die Möglichkeit, das Video-RAM innerhalb seines Adressbereiches von 16 KB zu verschieben. Der Speicherbereich, den der Grafikchip dazu verwendet, um den Inhalt des Textbildschirms aufzubauen muß also nicht zwingenderweise bei $0400 liegen, wo er sonst nach dem Einschalten des Rechners untergebracht ist. Durch die Bits 4-7 des Registers $D018 können insgesamt 16 verschiedene Speicherbereiche gewählt werden, deren Basisadressen in $0400-Schritten von $0000-$3C00 gehen.
Nun werden Sie fragen, was denn das Video-RAM mit unserer ESCOS-Routine zu tun hat, zumal wir die Bildschirmdarstellung doch sowieso abgeschaltet haben, und keinen Text auf dem Bildschirm haben?
Nun, ganz einfach: wo befinden sich denn die Register der Spritepointer normalerweise? Natürlich in den Adressen von $07F8-$07FF. Und genau diese Adresse liegen am Ende des Video-RAMs. Verschiebt man nun das Video-RAM an eine andere Adresse, so verschiebt man automatisch auch die Registeradressen der Spritepointer! Wird das Video-RAM also beispielsweise um $0400 Bytes nach $0800 verschoben, so bewegen sich die Spritepointer-Register ebenfalls um $0400-Bytes nach vorne. Um nun also das Aussehen der acht Sprites zu definieren, müssen die Adressen $0BF8-$0BFF beschrieben werden. Und genau das ist die Lösung unseres Problems. Da es während des Spriteaufbaus zu lange dauert, alle acht Spritepointer in den Akku zu laden und in die Register zu schreiben, soll das die Initialisierungsroutine des Moviescrollers übernehmen. Hierbei schreibt letztere die benötigten Werte für eine Spritezeile in die Pointerregister eines verschobenen Video-RAMs. Während der Darstellung brauchen wir nun nur noch durch Schreiben eines einzigen Bytes, nämlich des entsprechenden Wertes für ein neues Video-RAM in $D018, auf selbiges umzuschalten. Dadurch, daß der VIC für die neue Spritezeile nun in einem ganz anderen Video-RAM arbeitet, holt er sich auch die Spritepointer aus den neuen Adressen, womit wir alle acht Spritepointer mit nur einem Schreibzugriff umgeschaltet hätten!
Die Umsetzung dieser Lösung in die Praxis gestaltet sich für uns sehr einfach. Sinnigerweise haben wir nämlich in den Routinen "ESCOS1" und "ESCOS2" schon Platz für diese Änderung gelassen. Wie Sie sich vielleicht erinnern, hatten wir zur Darstellung der Sprites innerhalb der IRQ-Routine dieser beiden Beispielprogramme die Unterroutine " OPEN21" vierzehnmal aufgerufen. Sie wird immer in der ersten Rasterzeile einer neuen Spritezeile angesprungen, und übernimmt das Neusetzen der Y-Koordinaten aller Sprites, sowie das Öffnen des Sideborders in den folgenden 21 Rasterzeilen.
So sah der Aufruf in den beiden Beispielprogrammen aus:

```
                NOP             ;Diese Befehlsfolge wird
                JSR open21      ;insgesamt 14 x aufgerufen
                ...             ;um je 21 Zeilen zu öffnen
                NOP             ;Sprite Line 14
                JSR open21
```

Die "NOP"-Befehle dienten dabei nur der Verzögerung um zwei Taktzyklen, damit die Änderung zum richtigen Zeitpunkt eintritt. Wir können diese Befehle mit jedem anderen Befehl ersetzen, der nur 2 Takte in Anspruch nimmt. Diese Tatsache machen wir uns für den Moviescroller zunutze. Wir ersetzen die NOPs mit LDA-Befehlen, wobei der Wert, der in den Akku geladen wird, dem Wert entspricht, der in Register $D018 geschrieben werden muß, um auf das neue Video-RAM umzuschalten. Er dient quasi als Parameter für die "OPEN21"- Routine. Demnach sieht der Aufruf dieser Routine, aus dem Raster-IRQ heraus, nun folgendermaßen aus:

```
v00             LDA #$00*$10    ;Code für Scr0 ($0000)
                JSR open21      ;21 Rasterzeilen öffnen
v01             LDAlda #$01*$10 ;Code für Scr0 ($0400)
                JSR open21      ;21 Rasterzeilen öffnen
v02             LDA #$02*$10    ;Code für Scr0 ($0800)
                JSR open21      ;21 Rasterzeilen öffnen
                ...
v0d             LDA #$0D*$10    ;Code für Scr13 ($3400)
                JSR open21      ;21 Rasterzeilen öffnen
```

Wie schon in den ESCOS-Beispielen, so wiederholt sich auch hier die LDA-JSR-Befehlsfolge 14 Mal, wobei jeweils der nächste Video-RAM- Bereich als Parameter im Akku übergeben wird. Timingmässig hat sich nichts geändert, da der NOP-Befehl genau solange braucht wie der LDA-Befehl.
Eine ähnliche Modifikation haben wir nun für das Schreiben dieses Akku-Wertes in Register $D018 vorgenommen. Diese Aufgabe soll von der Routine "OPEN21" durchgeführt werden. Hier ein Auszug der ersten Zeilen dieser Routine aus dem Beispiel "ESCOS2" :

```
open21          NOP             ;Verzögern bis rechter
                NOP             ;Rand erreicht
                DEC $D016       ;38 Spalten (Rand
                INC $D016       ;40 Spalten öffnen)
                LDA $1500,Y     ;$D011-Wert aus Tabelle
                STA $D011       ;lesen und eintragen
                INY             ;Tabellen-Index+1
                JSR cycles+5    ;24 Zyklen verzögern
                …
```

Wie Sie sehen, stehen hier ebenfalls zwei NOP-Befehle am Anfang der Routine.
Sie benötigen 4 Taktzyklen, was für einen "STA $XXXX"- Befehl ebenfalls zutrifft. Die zwei NOPs wurden für den Moviescroller nun mit einem "STA $D018" ersetzt:

```
open21          STA $D018       ;VRAM f. SprPtrs. versch.
                DEC $D016       ;38 Spalten (Rand
                INC $D016       ;40 Spalten öffnen)
                LDA $1500,Y     ;$D011-Wert aus Tabelle
                STA $D011; lesen und eintragen
                INY             ;Tabellen-Index+1
                JSR cycles+5    ;24 Zyklen verzögern
                …
```

Damit hätten wir also die ESCOS-Routine so umprogrammiert, daß Sie uns in jeder der 14 Spritezeilen auch neue Spritepointer setzt. Es müssen nun noch zwei weitere Änderungen gemacht werden, damit der Moviescroller auch voll funktionstüchtig ist.
Zunächst einmal muß die Initialierungsroutine erweitert werden. Sie soll uns die Spritepointer der benutzten Video-RAM- Adressen auf bestimmte Sprites vorinitialisieren, so daß später zur Darstellung einer bestimmten Spritereihe nur noch die Nummer des zu benutzenden Video-RAMs in den Labeln "V00" bis "V0D" der Raster-IRQ- Routine (siehe oben) eingetragen werden muß, und die Routine somit automatisch den richtigen Bildschirm zur korrekten Darstellung der entsprechenden Spritezeile wählt.
Zunächst einmal wollen wir vereinbaren, daß wir den 16 K-Adressbereich des VICs von $0000-$3FFF um eins nach oben in den Bereich von $4000-$7FFF verschieben. Dadurch stört uns die Zeropage, die normalerweise ja auch im Bereich des VICs liegt, nicht mehr, und wir haben volle 16 KB zur Speicherung von Sprites und Spritepointer-Video-RAM zur Verfügung.
Die Verschiebung wird durch Schreiben des Wertes 2 in das Portregister A von CIA-B erreicht. Innerhalb der Initialisierung wurde also folgende Befehlssequenz hinzugefügt:

```
LDA #$02
STA $DD00
```

Damit befindet sich der Datenbereich für den VIC nun im Bereich von $4000-$7FFF. Beachten Sie bitte auch, daß nun der Bytewert, den der VIC in "ausgetricksten" Rasterzeilen darstellt, nicht mehr in $3FFF, sondern in $7FFF abgelegt werden muß (im Beispielprogramm enthält er den Wert $81, womit vertikale Linien hinter dem Scrolltext erscheinen). Nun folgt der Teil, der die Spritepointer setzt. Hier treffen wir die Konvention, daß die Sprites, die durch die Pointer einer Video-RAM- Bereichs dargestellt werden auch innerhalb dieses Video-RAMs unterbracht werden sollen. Liegt dieses also Beispielsweise bei $4400, so soll Spritepointer 0 auf das Sprite bei $4400 (Blocknr.16), Spritepointer 1 auf das Sprite bei $4440 (Blocknr.17), usw., zeigen. Dies bewerkstelligt nun der folgende Teil der Init-Routine:

```
LDA #$F8            ;ZP-Zgr. $02/03 mit
STA $02             ;$43F8 init.
LDA #$43
STA $03
LDA #($0000/64)     ;Ptr. f. Sprite-Line01
JSR setpoint        ;setzen
LDA #($0400/64)     ;Ptr. f. Sprite-Line02
JSR setpoint        ;setzen
LDA #($0800/64)     ; Ptr. f. Sprite-Line03
JSR setpoint        ;setzen
…
LDA #($3000/64)     ;Ptr. f. Sprite Line13
JSR setpoint        ; setzen
LDA #($3400/64)     ;Ptr. f. Sprite Line14
JSR setpoint        ;setzen
```

Wie Sie sehen wird lediglich ein Adressierungszeiger in der Zeropage initialisiert, und dann 14 Mal die Unterroutine "SETPOINT" aufgerufen, wobei im Akku der Inhalt für den jeweils ersten Spritepointer übergeben wird. Hier nun die Routine "SETPOINT", die die eigentlichen Werte in die Pointerregister schreibt:

```
SETPOINT    LDY #$00        ;Index-Register initialisieren
            SAT ($02),Y     ; SprPtr0 ablegen
```

```
                CLC             ;Akku=Akku+1
                ADC #$01
                INY             ;Index=Index+1
                STA ($02),Y     ;SprPtr1 ablegen
                CLC             ;Akku=Akku+1
                ADC #$01
                INY             ;Index=Index+1
                STA ($02),Y     ;SprPtr2 ablegen
                ...
                CLC             ;Akku=Akku+1
                ADC #$01
                INY             ;Index=Index+1
                STA ($02),Y     ;SprPtr7 ablegen
                CLC             ;Auf Hi-Byte des $02/$03
                LDA $03         ;Zeigers den Wert 4 addieren
                ADC #$04        ;um auf nächstes VRAM zu
                STA $03         ;positonieren
                RTS
```

Wie Sie sehen, so wird vom Basiswert des ersten Spritepointers an, acht Mal jeweils um eins hochgezählt und das Ergebnis über den Vektor bei $02/$03 in die entsprechende Sprite-Pointer-Adresse geschrieben. Beim ersten Durchlauf zeigt dieser Vektor auf Adresse $43F8, wo sich die Spritepointer des ersten Video-RAM- Bereichs befinden. Am Ende der Pointerinitialisierung wird die Vektoradresse nun um $0400 erhöht (auf Hi-Byte wird $04 addiert), damit beim nächsten Durchlauf die Zeiger des nächsten Video-RAM-Bereichs gesetzt werden.

Nachdem nun die Spritepointer vorinitialisiert wurden, muß die "MOVESPR"- Routine, die Sie bestimmt noch von unseren alten Moviescroller-Routinen her kennen, so modifiziert werden, daß sie die IRQ-Routine derart ändert, daß in jeder Spritezeile auch der richtige Video-RAM-Bereich eingeschaltet wird. Da die "MOVESPR"- Routine schon im 10. Teil des IRQ-Kurses ausführlich beschrieben wurde soll hier nur das kommentierte Listing die Funktionsweise der Routine wieder ins Gedächtnis rufen. Der eigentliche, für uns wichtige Teil der Routine befindet sich am Ende und heißt "ROLLON". Er wurde der ESCOS-Routine angepasst und soll anschließend aufgeführt werden. Zunächst jedoch die MOVESPR-Routine:

```
movespr         LDA softroll        ;Scroll-Offs. Holen
                SBC #$01            ;Und 1 subtrahieren
                STA softroll        ;neuen Scroll-Offs. abl.
                BPL rollon          ;Wenn >0, dann weiter
newlineINC      showline            ;1. Spr-Zeilen- Ind. Erh.
                SEC                 ;C-Bit f. Subtr. Setzen
                LDA showline        ;Showline holen
                SBC #$0D            ;und 13 subtr.
                BMI noloop          ;Bei Unterlauf weiter
                STA     showline    ;Sonst Wert abl.
noloop:         LDA tpoint
                CMP #<estext+($34*$18)
                BNE continue
                LDA tpoint+1
                CMP #<estext+($34*$18)
                BNE continue
                LDA #$00
                STA showline
                LDA #<estext
                STA tpoint+0
```

```
                LDA #>estext
                STA tpoint+1
continue        CLC             ;C-Bit f.Add. löschen
                LDA softroll    ;SOFTROLL laden
                ADC #$18        ;24 addieren
                STA softroll    ;und wieder ablegen
                LDA #$20        ;Op-Code für "JSR"
                STA mt          ;in MT eintragen
```

Ab dem Label "NEWLINE" ist die Routine damit beschäftigt, die neue Spritezeile aufzubauen, sowie die Zeiger für diese Zeile neu zu setzen. Nach ihrer Abarbeitung gelangt sie automatisch zum Teil "ROLLON", zu dem auch am Anfang verzweigt wird, wenn keine neue Spritezeile eingefügt werden muß. Hier geschehen nun die für uns wesentlichen Dinge, nämlich das Voreinstellen der richtigen Video-RAM-Werte innerhalb der IRQ-Routine.
Zunächst das Listing:

```
rollon          LDA softroll        ;nächsten Raster
                STA $D012           ;IRQ vorbereiten
                LDY showline        ;SprZeilen-Ind. Holen
                LDA pointer+$00,Y   ;Wert für.1. SprZeile
                STA v00,+1          ;in IRQ setzen
                LDA pointer+$01,Y   ;Wert für 2. SprZeile
                STA v01,+1          ;in IRQ setzen
                …
                LDA pointer+$0D,Y   ;Wert für14. SprZeile
                STA v0D,+1          ;in IRQ setzen
                RTS
```

Alles in Allem also keine schwierige Aufgabe. Zur Erinnerung sollte ich noch erwähnen, daß die beiden Labels "SOFTROLL" und "SHOWLINE" für die Zeropageadressen $F8 und $F9 stehen."SOFTROLL" ist dabei ein Zähler für den Scrolloffset, der pro Bildschirmdurchlauf einmal erniedrigt wird. Dieser Zähler gibt gleichzeitig auch die Rasterzeile an, an der der nächste Raster-IRQ auftreten muß. Dadurch, daß dieser Wert am Anfang der "MOVESPR"-Routine um eins erniedrigt wurde, und er hier nun als Auslöser für die nächste Rasterzeile eingetragen wird, erreichen wir den Scrolleffekt, da die Spritezeilen pro Rasterdurchlauf immer 1 Rasterzeile früher dargestellt werden, solange bis "SOFTROLL" als Zähler einen Unterlauf meldet, und wieder auf 21 zurückgesetzt wird. Gleichzeitig muß dann die soeben weggescrollte Spritezeile am unteren Bildschirmrand, mit neuem Text wieder eingefügt werden, was vom Abschnitt "NEWLINE" schon durchgeführt wurde."SHOWLINE" ist ein Zeiger auf die Tabelle "POINTER". Er gibt an, welche der 14 Spritezeilen momentan an der obersten Position steht. Er wird bei einem "SOFTROLL"- Unterlauf um eins hochgezählt, da nun ja die ehemals zweite Spritezeile zur Ersten wird. Damit tut unser neuer "ROLLON"- Teil nichts anderes, als für jede Spritezeile einen Wert aus der Tabelle zu lesen und in eins der Labels von "V00" bis "V0D" einzutragen.
Wie aus dem obigen Auszug der IRQ-Routine ersichtlich, so ist dies jeweils der LDA-Befehl, der vor jedem "JSR OPEN21" steht. Da wir auf die Labeladressen den Wert 1 addieren, schreiben wir also einfach einen neuen Operanden für den LDA-Befehl in das Programm, so daß die IRQ-Routine automatisch den richtigen Pointerwert übergibt. Die Pointertabelle sieht nun wie folgt aus:

```
POINTER:    .byte $00,$10,$20,$30
            .byte $40,$50,$60,$70
            .byte $80,$90,$A0,$B0
```

```
.byte $C0,$D0
.byte $00,$10,$20,$30
.byte $40,$50,$60,$70
.byte $80,$90,$A0,$B0
.byte $C0,$D0
```

Wie Sie sehen, enthält sie einfach die benötigten Werte für das VIC-Register $D018, um der Reihe nach die Video-RAM-Bereiche zwischen $4000 und $7400 einzuschalten. Die Tabelle enthält, wie auch schon bei den alten Moviescroller-Routinen, alle Pointerdaten doppelt, damit "SHOWLINE" auch bis zum Wert 13 hochgezählt werden kann, und dennoch die Werte korrekt gesetzt werden.
Damit wären nun alle relevanten Teile der Änderung einer ESCOS-Routine zum Moviescroller erklärt worden. Wie Sie sehen, haben wir auf recht einfach Art und Weise zwei verschiedene Rastereffekte miteinander kombiniert, ohne große Änderungen vornehmen zu müssen. Das Ergebnis ist quasi ein Moviescroller mit ESCOS-IRQ-Routine. Auf diese Weise ist es auch möglich ganz andere Effekte miteinander zu kombinieren, was Ihre Phantasie bezüglich der Raster-IRQ- Programmierung ein wenig beflügeln soll.
Das hier besprochene Programmbeispiel befindet sich wie immer auch auf dieser MD und kann von Ihnen begutachtet werden. Es heißt "MOVIE V1 .2" und muß mit ",8,1" geladen und durch ein " SYS4096" gestartet werden. Im nächsten Kursteil werden wir dann die, schon versprochenen, Routinen zum Dehnen von Sprites besprechen.

(ih/ub)

Teil 13 – Magic Disk 11/94

Herzlich Willkommen zum dreizehnten Teil unseres IRQ-Kurses. Auch diesen Monat soll es um das trickreiche manipulieren von Sprites gehen. Wir werden uns einige Routinen zum Dehnen von Sprites anschauen, und dabei lernen, daß es auch Sprites gibt, die mehr als 21 Rasterzeilen hoch sind…

1. Das Prinzip

Wie immer kommen wir zu Beginn zum Funktionsprinzip des Rastereffektes dieses Kursteils: Wie Sie vielleicht wissen, so ist die Größe eines Sprites prinzipiell auf 24x21 Pixel begrenzt. Diese Größe ist starr und eigentlich nicht veränderbar. Es existieren jedoch 3 Sonderfälle, in denen das Sprite auch größer sein kann, nämlich dann, wenn wir mit Hilfe der Register $D017 und $D01D die X und Y-Expansion eines Sprites einschalten. In diesem Fall kann sowohl die X- als auch die Y-Ausdehnung verdoppelt werden, so daß wir ein Sprite mit einer maximalen Größe von 48x42 Pixeln erhalten, in dem die normalen 24x21 Pixel einfach doppelt dargestellt werden. Wie Sie also sehen ist der VIC rein theoretisch doch in der Lage größere Sprites auf den Bildschirm zu zaubern. Jedoch auch dies nur in höchst beschränkter Form, da wir nun ebenfalls an eine fixe Auflösung gebunden sind. Wir werden allerdings gleich ein Verfahren kennenlernen, mit dem es uns möglich sein wird, zumindest die Y-Auflösung eines Sprites variabel festzulegen.
Drehund Angelpunkt dieses Effektes wird das eben schon angesprochene Register zur Y-Expansion der Sprites ($D017) sein. Wollen wir zunächst einmal klären, wie der VIC die Sprites überhaupt darstellt: Nehmen wir also an, daß wir ein Sprite auf dem Bildschirm darstellen möchten. Zunächst einmal nicht expandiert. Der VIC vergleicht nun jede Rasterzeilennummer mit der Y-Position des Sprites. Sind beide Werte gleich, so hat er die Rasterzeile erreicht, in der die oberste Linie des Sprites zu sehen sein soll. Es wird nun eine interne Schaltlogik aktiviert, die dem VIC zu Beginn einer jeden Rasterzeile die Adresse der nächsten drei Datenbytes an den Datenbus legt und ihn somit jedesmal mit

den in dieser Zeile für dieses Sprite relevanten Daten füttert. Die Schaltlogik verfügt nun für jedes Sprite über ein 1-Bit-Zähl-, sowie ein 1-Bit-Latch- Register, die beide bei einem Schreibzugriff auf das Register $D017, mit dem Zustand des Bits zum zugehörigen Sprite neu initialisiert werden. Das Latch-Register dient dabei als "Merkhilfe" für den Zustand der Y-Expansion des Sprites.
Enthält es den Bitwert 1, so soll das Sprite in Y-Richtung verdoppelt werden, enthält es den Wert 0, so soll es normal dargestellt werden. Ab der Rasterzeile, ab der das Sprite nun gezeichnet werden soll legt die Schaltlogik nun zunächst die aktuelle Spritedatenadresse an den Bus und füttert den VIC so mit den Spritedaten für die erste Rasterzeile.
Gleichzeitig wird bei Erreichen der nächsten Rasterzeile der 1-Bit-Zähler um eins erniedrigt. Tritt dabei ein Unterlauf auf, so reinitialisiert die Schaltlogik den Zähler wieder mit dem Inhalt des Latch-Registers und erhöht die Quelladresse für die Spritedaten um 3 Bytes, so daß Sie anschließend dem VIC die Daten der nächsten Spritezeile zukommen lässt. Tritt kein Unterlauf auf, weil der Zähler auf 1 stand und beim Herunterzählen auf 0 sprang, so bleibt die Quelladresse der Spritedaten unverändert, so daß der VIC auch in dieser Rasterzeile dieselben Spritedaten liest, die er auch eine Rasterzeile vorher schon darstellte. Ist die Spriteexpansion nun abgeschaltet, so wurden Latch und Zähler zu Beginn mit dem Wert 0 gefüttert. In dem Fall läuft der Zähler in jeder Rasterzeile unter und somit bekommt der VIC in jeder Rasterzeile neue Spritedaten zugewiesen. Ist die Y-Expansion eingeschaltet, so enthalten Zähler und Latch den Wert 1 und es wird fortlaufend nur in jeder zweiten Rasterzeile eine neue Adresse an den Datenbus angelegt, womit der VIC das Sprite in der Vertikalen doppelt so hoch darstellt.
Wie nun eingangs schon erwähnt so werden sowohl Latch als auch Zähler neu initialisiert, sobald ein Schreibzugriff auf Register $ D017 erfolgt. Dadurch haben wir also auch direkten Einfluß auf den Y-Expansions- Zähler. Was sollte uns nun also davon abhalten, diesen Zähler in JEDER Rasterzeile durch Setzen des Y-Expansionsbits des gewünschten Sprites wieder auf 1 zurückzusetzen, so daß die Schaltlogik nie einen Unterlauf erzeugen kann, und somit immer wieder dieselben Spritedaten angezeigt werden? Und ganz genau so können wir ein Sprite länger strecken als es eigentlich ist und z. B.
3-,4-, oder 5-fache Expansion des Sprites bewirken (indem jede Spritezeile 3-,4- oder 5-mal hintereinander dargestellt wird) ! Wir haben sogar die Möglichkeit jede Spritezeile beliebig, und voneinander unabhängig oft, zu wiederholen, so daß man z. B. auch eine Sinuswelle über das Sprite laufen lassen kann! Hierzu muß lediglich exakt zu Beginn einer Rasterzeile entweder das Expansionsbit gesetzt werden, wenn die Rasterzeile dieselben Spritedaten enthalten soll wie die letzte Rasterzeile, oder wir Löschen das Expansionsbit des gewünschten Sprites, um die Daten der nächsten Spritezeile in den VIC zu holen!

2. Programmbeispiele 1 und 2

Um einen Eindruck von den Möglichkeiten zu bekommen, die uns dieser Effekt bietet, sollten Sie sich einmal die Beispielprogramme "STRETCHER.1" bis "STRET-CHER.4" auf dieser MD anschauen. Sie werden alle wie immer mit LOAD"Name",8,1 geladen und durch ein "SYS4096" gestartet. Die ersten beiden Beispiele stellen ein Sprite dar, das normalerweise nur eine diagonale Line von der linken oberen Ecke zur rechten unteren Ecke des Sprites enthält. Im ersten Beispiel haben wir lediglich einige dieser Zeilen mehrfach dargestellt. Das zweite Beispiel enthält eine Streckungstabelle, mit der wir jede Rasterzeile in Folge 1-,2-,3-,4-,5-, und 6-Mal darstellen, womit die Linie in etwa die Rundungen einer Sinuskurve bekommt!
Wollen wir uns nun einmal den Programmcode anschauen, den wir zur Erzeugung der Verzerrung in Beispiel "STRETCHER.1" verwenden. Die Initialisierung und den Beginn der IRQ-Routine möchte ich wie immer aussparen, da beides absolut identisch mit unseren anderen IRQ-Beispielen ist. Wir legen hier den Raster-IRQ auf Rasterzeile $82 fest und

schalten Betriebssystem- ROM ab, um direkt über den IRQ-Vektor bei $FFFE/$FFFF zu springen.
Die IRQ-Routine selbst beginnt nun ab Adresse $1100, wo zunächst unser altbekannter Trick zum Glätten des IRQs aufgeführt ist. Der für uns wesentliche Teil beginnt wie immer ab dem Label "ONECYCLE", ab den der IRQ geglättet wurde, und die für uns relevanten Routinenteile stehen. Zusätzlich sei erwähnt, daß wir gleichzeitig, um Timingprobleme zu vermeiden, eine FLD-Routine benutzen um die Charakterzeilen wegzudrücken und gleichzeitig den linken und rechten Bildschirmrand öffnen, damit wir in späteren Beispielen auch Sprites in diesen Bereichen darstellen und sehen können.
Hier nun jedoch zunächst der Source-Code:

```
onecycle    LDA #$18          ;1. Wert für FLD
            STA $D011         ;in $D011 schreiben
            LDA #$F8          ;Nächsten IRQ bei Raster-
            STA $D012         ;zeile $F8 auslösen
            DEC $D019         ;VIC-ICR löschen
            LDA #21*5         ;Zähler f. FLD init. (21*5=
            STA $02           ;5-fache Spritehöhe)
            NOP               ;Verzögern..
            LDX #$00          ;Tabellenindex init.
fldloop     LDA ad017,X       ;Wert aus Stretch-Tab lesen
            STA $D017         ;und in Y-Exp. eintragen
            LDA ad011,X       ;Wert aus FLD-Tab lesen
            STA $D011         ;und in $D011 eintragen
            DEC $D016         ;38 Spalten (Rand
            INC $D016         ;40 Spalten öffnen)
            NOP               ;Bis zum Anfang der
            NOP               ;nächsten Rasterzeile
            NOP               ;verzögern...
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            LDA #$00          ;Y-Exp. auf 0
            STA $D017         ;zurücksetzen
            INX               ;Tab-Index+1
            CPX $02           ;Mit FLD-Zähler vergleichen
            BCC fldloop       ;Kleiner, also weiter
            LDA #$82          ;Nächster IRQ bei Rasterz.
            STA $D012         ; $82
            LDX #$00          ;IRQ-Vektoren
            LDY #$11          ;auf eigene
            STX $FFFE         ;Routine
            STY $FFFF         ;umstellen
            LDA #$0E          ;Farben auf hellblau/
            STA $D020         ;schwarz setzen
            LDA #$00
            STA $D021
            PLA               ;Prozessorregs. wieder vom
            TAY               ;Stapel holen und IRQ
            PLA               ;beenden
            TAX
            PLA
            RTI
```

Ab dem Label ONECYCLE setzen wir zunächst einige Basiswerte für die Sprite-Stretch,

FLD und Sideborderroutinen.
Hierzu schreiben wir erst einmal die Anzahl der Rasterzeilen, in denen diese drei Effekte aktiv sein sollen als Vergleichszähler in Adresse $02 und löschen das X-Register, das als Index auf die FLD und Sprite-Stretch-Tabellen dienen soll (dazu später mehr) . Das Setzen des nächsten IRQ-Auslösers auf Rasterzeile $F8 ist lediglich ein Relikt aus älteren IRQ-Routinen, in denen wir den unteren und oberen Rand des Bildschirms öffneten. Da wir später jedoch wieder Rasterzeile $82 als IRQ-Auslöser festlegen, ist diese Befehlsfolge eigentlich unnötig. Dennoch haben wir sie beibehalten, da das Entfernen der beiden Befehle zum Einen das Timing, das zum exakten synchronisieren zwischen Programm und Rasterstrahl notwendig ist, durcheinanderbrächte und wir dadurch andere Befehle zum Verzögern einfügen müssten, und zum Anderen um die Flexibilität der Routine dadurch nicht einzuschränken. Auf diese Weise wird es z.B. für Sie sehr einfach, die Routine mit einer Top- und Bottom-Border-Funktion "nachzurüsten", indem Sie lediglich die IRQ-Vektoren weiter unten auf eine solche Routine verbiegen und das Festlegen des nächsten IRQs bei Rasterzeile $82 auf diese Routine verlagern. Dies ist übrigens eine saubere Möglichkeit timingkritische IRQ-Routinen zu schreiben, und sie zusätzlich zu anderen Raster-Effekten erweiterbar zu halten. Da wir sowieso die meiste Zeit verzögern müssen tun uns die beiden Befehle auch nicht weiter weh.
Es folgt nun der eigentliche Kern der IRQ-Routine; eine Schleife namens "FLD-LOOP" . Hier lesen wir zunächst einmal einen Wert aus der Tabelle "AD017" aus und tragen ihn in das Y-Expansions- Register ein. Die besagte Tabelle befindet sich im Code ab Adresse $1600 und enthält die Werte, die nötig sind, um das Sprite wie gewünscht zu dehnen. Da wir uns in den Beispielen 1 und 2 nur auf ein Sprite beschränken (Sprite 0 nämlich), enthält die Tabelle natürlich nur $00- und $01-Werte. Bei $00 wird ganz normal die nächste Spritezeile gelesen und angezeigt, bei $01 wird immer wieder die zuletzt dargestellte Spritezeile auf den Bildschirm gebracht. Folgen mehrere $01-Werte aufeinander, so wird eine einzige Spritezeile so oft wiederholt, wie $01-Werte in der Tabelle stehen. Um die nächste Spritezeile darzustellen muß jetzt mindestens einmal ein $00-Wert folgen. Diese Zeile kann nun ebenfalls beliebig oft wiederholt werden, usw. Zur besseren Übersicht hier ein Auszug aus der Tabelle mit Ihren Werten für das Beispiel "STRETCHER.1":

```
ad017       .byte $01,$01,$01,$00,$01,$00,$00,$00
            .byte $00,$00,$00,$00,$00,$00,$00,$00
            .byte $01,$01,$01,$00,$01,$00,$00,$00
            .byte $00,$00,$00,$00,$00,$00,$00,$00
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$01,$01,$01,$01
            .byte $01,$01,$01,$01,$00,$00,$00,$00
```

Wie Sie sehen verzerren wir hier zunächst die ersten Spritezeilen verschieden oft. Hiernach wird die letzte Spritezeile so oft wiederholt, bis das Ende unseres, durch FLD und Sideborder behandelten, Bildschirmbereichs erreicht wurde.
Kommen wir jedoch wieder zu unserer FLD-LOOP zurück. Nach dem Setzen des Y-Expansions-Registers wird abermals ein Tabellenwert gelesen und diesmal in Register $D011 übertragen. Diese Befehlsfolge ist für den FLD-Effekt notwendig, mit dem wir den Beginn der nächsten Charakterzeile vor dem Rasterstrahl herschieben. Die Tabelle enthält

immer wieder die Werte $19, $1A, $1B, $1C, $1D, $1E, $1F, $18, usw., womit wir in jeder Rasterzeile die Horizontalverschiebung um eine Rasterzeile versetzt vor dem Rasterstrahl herdrücken.
Als Nächstes folgt das Öffnen des linken und rechten Bildschirmrandes durch die altbekannte Befehlsfolge zum schnellen Runter- und wieder Hochschalten zwischen 38- und 40-Spalten-Darstellung.
Durch die nun folgenden 9 NOP-Befehle verzögern wir solange, bis der Rasterstrahl eine Position erreicht hat, zu der die VIC-Schaltlogik schon die gewünschte Spritezeilenadresse an den Datenbus angelegt hat, und somit der VIC mit den gewünschten Spritedaten für diese Zeile gefüttert wurde. Jetzt schalten wir die Y-Expansion wiederum ganz ab, um das Register für den nächsten Wert vorzubereiten. Gleichzeitig stellen wir damit sicher, daß der Rest des Sprites, der ggf. über unseren, vom Raster-IRQ behandelten, Bereich hinausragt (z. B. wenn wir das Sprite zu lang gedehnt haben), in normaler Darstellung auf den Bildschirm gelangt. Es wird nun nur noch der Index-Zähler im X-Register für den nächsten Schleifendurchlauf um 1 erhöht und mit der Anzahl der Raster-Effekt-Zeilen in Register $02 verglichen. Ist er noch kleiner als dieser Wert, so können wir die Schleife wiederholen, um die nächste Rasterzeile zu bearbeiten. Im anderen Fall sind wir am Ende angelangt, wo der nächste Interrupt vorbereitet wird, bevor wir die IRQ-Routine wie gewohnt beenden.
Damit hätten wir auch schon den Kern unserer Routine besprochen. Experimentieren Sie doch ein wenig mit der Verzerrung, indem Sie die Tabelle "AD017" ab Adresse $1600 mit Hilfe eines Speichermoditors abändern. Sie werden sehen welch lustige Deformierungen dabei entstehen können! Beachten Sie dabei, daß Sie die Form des Sprites im Speicher niemals ändern, sonder daß die Änderung hardwaremäßig eintritt!

Kommen wir nun zum Beispiel "STRET-CHER.2" . Rein äußerlich unterscheidet sich diese Routine nicht von "STRET-CHER.1" . Auch hier wird dasselbe Sprite wieder verzerrt dargestellt, wobei die Verzerrung jedoch etwas anders ausfällt.
Rein theoretisch haben wir nur die Tabelle "AD017" verändert, so daß dasselbe Sprite ein anderes Aussehen erlangt.
Technisch gesehen wurde dieses Beispiel jedoch um ein zusätzliches Feature erweitert: Am Ende unserer IRQ-Routine, genau bevor wir die Prozessorregister wieder zurückholen und den IRQ mittels RTI verlassen, haben wir den Befehl "JSR $1300" hinzugefügt. An dieser Adresse befindet sich nun eine kleine Unterroutine, die es uns ermöglicht, flexiblere Dehnungen zu programmieren, ohne, daß wir uns Gedanken über den Aufbau der Tabelle "AD017" machen müssen. Sie greift auf eine Tabelle namens "LSTRETCH" zu, die 21 Bytes enthält, die jeweils die Anzahl der Rasterzeilen angeben, die eine jede der 21 Spritezeilen wiederholt werden soll. Die Tabelle liegt ab Adresse $1700 und sieht folgendermaßen aus:

```
lstretch .byte $01          ;Spriteline01 1x Wdh.
         .byte $02          ;Spriteline02 2x Wdh.
         .byte $03          ;Spriteline03 3x Wdh.
         .byte $04          ;Spriteline04 4x Wdh.
         .byte $05          ;Spriteline05 5x Wdh.
         .byte $06          ;Spriteline06 6x Wdh.
         .byte $06          ;Spriteline07 6x Wdh.
         .byte $05          ;Spriteline08 5x Wdh.
         .byte $04          ;Spriteline09 4x Wdh.
         .byte $03          ;Spriteline10 3x Wdh.
         .byte $02          ;Spriteline11 2x Wdh.
         .byte $01          ;Spriteline12 1x Wdh.
         .byte $01          ;Spriteline13 1x Wdh.
         .byte $01          ;Spriteline14 1x Wdh.
```

```
.byte $01       ;Spriteline15 1x Wdh.
.byte $01       ;Spriteline16 1x Wdh.
.byte $01       ;Spriteline17 1x Wdh.
.byte $01       ;Spriteline18 1x Wdh.
.byte $01       ;Spriteline19 1x Wdh.
.byte $01       ;Spriteline20 1x Wdh.
.byte $00       ;Spriteline21 0x Wdh.
```

Die Routine bei $1300("STCHART" ist ihr Name) soll nun diese Werte in Folgen von $00/$01-Bytes umrechnen, und in der Tabelle "AD017" ablegen, so daß wir lediglich angeben müssen, welche Spritezeile wie oft wiederholt werden soll. Hier nun der Sourcecode der STCHART-Routine:

```
stchart     LDX #20         ;Zähler in X-Reg. initialisieren
            LDA #$FF        ;Akku mit $FF initialisieren
fill        STA ad017,X     ;AD017-Tab mit $FF
            INX             ;auffüllen (=letzte
            CPX #21*5       ; Spritezeile immer bis
            BNE fill        ; Ende wiederholen)
            LDY #$00        ;Index f. LSTRETCH-Tab
            LDX #$00        ;Index f. AD017- Tab nextline:
            LDA lstretch,Y  ;1 . Wert lesen und in
            STA $FF         ;$ FF ablegen
double      DEC $FF         ;Wert-1
            BEQ normal      ;Wert=0-> nächst. Zeile
            LDA #$01        ;Sonst 1 x wiederholen in
            STA ad017,X     ; AD017- Tab eintragen
            INX             ;AD017- Index+1
            JMP double      ;Und nochmal durchlaufen normal:
            LDA #$00        ;Code für "nächste Zeile
            STA ad017,X     ;lesen" in AD017-Tab
            INX             ;AD017-Index+1
            INY             ;LSTRETCH-Index+1
            CPY #20         ;Mit Ende vgleichen
            BNE nextline    ;Nein, also nochmal
            RTS             ;Sonst Ende
```

Wie Sie sehen füllt diese Routine lediglich die AD017- Tabelle so oft mit $01- Werten, wie der Bytewert einer Spritezeile aus LSTRETCH groß ist. Auf diese Weise können wir die Verzerrung einfacher handhaben, was uns in Beispiel 2 noch nichts nutzt, da hier immer nur dieselbe LSTRETCH-Tabelle benutzt wird, was uns aber bei den Beispielen 3 und 4 sehr zugute kommt.

3. Programmbeispiele 3 und 4

Diese beiden Beispiele bauen nun auf den Grundstein, den wir mit den ersten beiden Programmen legten, auf."STRETCHER.3" ist ein Programm, in dem Sie einen kleinen Flugsaurier auf dem Bildschirm flattern sehen. Diesen können Sie nun mit Hilfe eines Joysticks in Port 2 nach links und rechts über den Bildschirm bewegen. Drücken Sie den Joystick jedoch nach oben und unten, so können Sie unseren kleinen Freund wachsen oder wieder schrumpfen lassen, also fließend auf 5- fache Größe dehnen und wieder auf seine Ursprungsgröße zusammenschrumpfen lassen. Hierzu haben wir lediglich eine Joystickabfrage miteingebaut, die bei nach unten gedrücktem Joystick einen der Werte aus der Tabelle LSTRETCH um 1 erhöht, bei nach oben gedrücktem Joystick einen dieser Werte erniedrigt.
Dadurch, daß nun gleichzeitig auch nach jedem Rasterdurchlauf die Routine STCHART aufgerufen wird, wird somit die kleine Änderung durch die Joystickroutine direkt in die

AD017-Tabelle übertragen, womit ein quasi stufenloser Dehn- und Staucheffekt entsteht. Hierbei ist es übrigens wichtig, die richtige Spritezeile für eine Vergrößerung zu wählen. In der Regel nimmt man dazu ein iteratives Verfahren, das die Anzahl der Spritezeilen, die eine Rasterzeile mehr dargestellt werden als Andere, so gleichmäßig verteilt, daß der Dehneffekt besonders flüssig erscheint. Dies sind jedoch Feinheiten, die wir hier nicht weiter besprechen möchten, da sie nicht zum Thema gehören.
Das Beispiel "STRETCHER.4" ist nun eine weitere Kombination aus den Beispielen 2 und 3. Wir haben hier acht Mal den kleinen Flugsaurier auf dem Bildschirm, der in Sinuswellenform quasi über den Bildschirm "schwabbelt". Hierbei haben wir uns eines einfachen Tricks bedient: Wir nahmen zunächst die LSTRETCH-Tabelle aus "STRETCHER.2", die eine Art runde Verzerrung in Vertikalrichtung erzeugt.
Diese Tabelle wird nun zyklisch durchgerollt. Pro Rasterdurchlauf kopieren wir die Bytes 0-19 um ein Byte nach vorne und setzen den Wert von Byte 20 wieder bei 0 ein. Dadurch entsteht der Effekt, als würden die kleinen Saurier als Reflektion auf einer gewellten Wasseroberfläche sichtbar sein!
Diese einfache Variation zeigt, mit welch simplen Methoden man eindrucksvolle Effekte durch Spritedehnung erzielen kann. Sie soll Sie wieder anregen, eigene Experimente mit unserer neuen Raster-Effekt-Routine durchzuführen.

(ih/ub)

Teil 14 – Magic Disk 12/94

Unser Kurs neigt sich langsam dem Ende zu, und als wahre " Raster-Feinschmecker" haben wir uns das beste Stückchen Code ganz für den Schluß aufgehoben. In diesem und den nächsten beiden (letzten) Kursteilen werden wir uns mit Raster-Tricks beschäftigen, die es uns ermöglichen, den Bildschirm des C64 HARDWAREMÄSSIG in alle Richtungen zu scrollen.
"Hardwaremässig" heißt, daß wir nicht etwa die Softscroll-Register des VICs beschreiben, und dann alle 8 Scollzeilen/- spalten den gesamten Bildschirm um 8 Pixel (oder einen Charakter) weiter kopieren, sondern daß wir vielmehr den VIC derart austricksen, daß er uns diese Arbeit von alleine abnimmt, und zwar ohne, daß wir auch nur einen einzigen Taktzyklus zum Kopieren von Grafikdaten verschwenden müssen! Die hierzu notwendigen Routinen heißen "VSP", zum vertikalen Scrollen,"HSP", zum horizontalen Scrollen, sowie "AGSP", die eine Kombination Kombination aus den beiden ersteren darstellt. Mit ihnen können wir den gesamten Bildschirm ganz problemlos (auch Hires-Grafiken!) in alle Richtungen verschieben. Im heutigen Kursteil wollen wir mit der VSP-Routine beginnen:

1. FLD und VSP - Die (un)gleichen Brüder

Sie werden sich jetzt sicher wundern, warum hier der Begriff "FLD", auftaucht, war das doch eine der "einfacheren" Routinen, die wir schon zu Anfang dieses Kurses besprochen hatten. Doch wie ich schon öfter erwähnte, ist die FLD-Routine meist der Schlüssel zu den komplexeren Rastertricks, und leistete uns auch schon manchen guten Dienst um Timingprobleme extrem zu vereinfachen.
Diesmal jedoch dreht sich alles direkt um unseren kleinen Helfershelfer, da die VSP-Routine sehr stark mit ihm verwandt ist, wenn die Beiden nicht sogar identisch sind."VSP" steht für "Vertical Screen Position", was der einfach Ausdruck für das ist, was die Routine bewirkt: durch sie wird es uns ermöglicht, den gesamten Bildschirm ohne jeglichen Kopieraufwand vollkommen frei nach oben oder unten zu scrollen. Und jetzt der Clou an der ganzen Sache: FLD und VSP unterscheiden sich lediglich durch einen einzigen NOP-Befehl voneinander. Fügt man Letzterern den Verzögerungs-NOPs nach der IRQ-Glättung der FLD-Routine hinzu, so erhält man eine voll funktionstüchtige VSP-Routine, die

gleichzeitig noch einen FLD-Effekt miteingebaut hat. Damit Sie genau wissen, wovon wir hier sprechen, sollten Sie sich auf dieser MD einmal die Programmbeispiele "FLD" und "VSP1" anschauen. Ersteres ist die FLD-Routine, so wie wir sie zu Beginn dieses Kurses kennengelernt hatten.
Durch Bewegen des Joysticks nach oben und unten können wir mit ihr den Bildschirm nach unten "wegdrücken". Das Prinzip, das dabei verfolgt wurde war, daß die FLD-Routine den Beginn der nächsten Charakterzeile vor dem Rasterstrahl herschob, indem sie ihn ständig mit neuen vertikalen Verschiebeoffsets in Register $D011 fütterte. Da er deshalb glaubte, sich noch nicht in der richtigen Rasterzeile zu befinden, in der er die nächste Charakterzeile zu lesen hatte,"vergaß" er solange sie zu zeichnen, bis wir ihm durch Beenden der FLD-Schleife die Möglichkeit dazu gaben, endlich die Rasterzeile zu erreichen, in der er nun tatsächlich die versäumte Charakterzeile lesen und anzeigen durfte. Egal, wieviele Rasterzeilen wir ihn dadurch "vertrödeln" ließen, er begann dann immer bei der Charakterzeile, die er eigentlich als nächstes aufzubauen gehabt hätte, so als wenn der FLD-Effekt nie aufgetreten wäre. War das die erste Charakterzeile des Bildschirms, so konnte man auch problemlos den Bildschirm erst in der Mitte oder am unteren Bildschirmrand beginnen lassen (so arbeitet übrigens auch der Effekt, mit dem Sie die Seiten, die Sie gerade lesen umblättern).

2. Das Programmbeispiel VSP1
Wollen wir uns nun einmal die IRQ-Routine des Programmbeispiels "VSP1" anschauen. Im Prinzip nichts besonderes, da sie, wie schon erwähnt, fast identisch mit der FLD-Routine ist. Sie ist ab Adresse $1100 zu finden und wird wie die meisten unserer IRQ-Routinen von der Border-IRQ-Routine, die wir immer zum Abschalten des unteren und oberen Bildrandes benutzen initialisiert:

```
fld     PHA             ;Prozessorregister
        TXA             ;auf Stapel retten
        PHA
        TYA
        PHA
        DEC $D019       ;VIC-ICR löschen
        INC $D012       ;Glättungs-IRQ
        LDA #<irq2      ;vorbereitem
        STA $FFFE
        CLI             ;IRQs freigeben
ch      NOP             ;Insgesamt 13 NOPs
        ...             ;zum Verzögern bis
        NOP             ;zum nächsten
        JMP ch          ;Raster-IRQ
irq2    PLA             ;Statusregister und
        PLA             ;IRQ-Adresse vom
        PLA             ;Stapel werfen
        DEC $D019       ;VIC-ICR freigeben
        LDA #$F8        ;Rasterpos. f. Bor-
        STA $D012       ;der IRQ festlegen
        LDX #<bord      ;IRQ-Vektoren
        LDY #>bord      ;auf Border-IRQ
        STX $FFFE       ;zurücksetzen (für
        STY $FFFF       ;nächsten Durchlauf)
        NOP             ;Verzögern bis
        NOP             ;zum Raster-
        NOP             ;zeilenende
        NOP
        NOP
        NOP
```

```
                NOP
                LDA $D012       ;den letzen Cyclus
                CMP $D012       ;korrigieren
                BNE onecycle
onecycle        LDA #$18        ;25-Zeilen-Bildschirm
                STA $D011       ;einschalten
                NOP             ;Insgesamt 16 NOPs zum
                …               ;Verzögern für FLD
```

Es folgt nun der alles entscheidende siebzehnte NOP-Befehl, der den VSP-Effekt auslöst:

```
                NOP             ;NOP für VSP
```

Nun geht's weiter mit dem normalen FLD-Code. Entfernen Sie das obige NOP aus dem Code, so erhalten Sie wieder eine ganz normale FLD-Routine!

```
                LDA $02         ;Zeilenz. holen
                BEQ fldend      ;Wenn 0 -> Ende
                LDX #$00        ;Zeiger inititialisieren
fldloop         LDA rollvsp,X   ;FLD-Offset zum
                ORA #$18        ;Verschieben der
                STA $D011       ;Charakterzeile
                LDA colors1,X   ;Blaue Raster
                STA $D020       ;im FLD-Be-
                STA $D021       ;reich darstellen
                NOP             ;Bis Zeilen-
                NOP             ;mitte verzö-
                NOP             ;gern
                NOP
                NOP
                NOP
                NOP
                LDA colors2,X   ;Rote Raster
                STA $D020       ;Im FLD-Be-
                STA $D021       ;reich darstellen
                NOP             ;Verzögern bis
                NOP             ;Zeilenende
                BIT $EA
                INX             ;Zeilenz.+1
                CPX $02         ;Mit $02 vgleichen
                BCC fldloop     ;ungl.->weiter
fldend          NOP             ;Verzögerung bis
                NOP             ;Zeilenende
                NOP
                NOP
                NOP
                LDA #$0E        ;Normale
                STA $D020       ;Bildschirm-
                LDA #$06        ;farben ein-
                STA $D021       ;schalten
                PLA             ;Prozessorregs. wieder vom
                TAY             ;Stapel holen und IRQ
                PLA             ;beenden
                TAX
                PLA
                RTI             ;und Ende
```

Die Tabelle "ROLLVSP" enthält Softscroll-Werte für Register $D011(nur die untersten 3 Bit sind genutzt), die die vertikale Verschiebung immer um eine Rasterzeile vor dem Rasterstrahl herschieben, so daß der FLD-Effekt entstehen kann.
Die Tabellen "Color1" und "Color2" geben die Farben an, die die FLD-Routine im

weggedrückten FLD-Bereich darstellt.
Dies ist nur als "Verschönerung" nebenbei gedacht.

3. Das Funktionsprinzip von VSP

Intern kann man sich die Vorgehensweise des VIC beim Aufbauen des Bildschirms mit all seinen Rasterzeilen folgendermaßen vorstellen: Trifft unser armer Grafikchip auf eine Rasterzeile, in der eigentlich die nächste Charakterzeile erscheinen sollte, so bereitet er sich auf den gleich folgenden Zugriff auf das Video-RAM vor, und zählt schon einmal einen internen Adresszeiger auf die gewünschte Adresse um 40 Zeichen nach oben, um die folgenden 40 Bytes rechtzeitig lesen zu können. Nun vergleicht er die Rasterstrahlposition ständig mit seiner Startposition für den Lesevorgang und hält beim Erreichen von Selbiger den Prozessor für 42 Taktzyklen an, um seinen Lesezugriff durchzuführen. Durch die FLD-Routine wurde nun jedoch der Beginn der gesamten Charakterzeile ständig vor dem VIC hergeschoben, weswegen er sie auch schön brav erst später zeichnete.
Unsere VSP-Routine verfügt nun über einen einzigen NOP mehr, der hinter der IRQ-Glättung eingefügt wurde. Das bedeutet, daß der FLD-Schreibzugriff auf Register $D011, mit dem der vertikale Verschiebeoffset um eins erhöht wird, exakt 2 Taktzyklen später eintritt als sonst.
In genau diesen beiden Taktzyklen aber hat der VIC schon die interne Adresse auf das Video-RAM um 40 Zeichen erhöht und wartet jetzt auf das Erreichen der Startposition für den Lesevorgang. Da der in unserem Programm folgende Schreibzugriff auf $D011 diese Position für den VIC nun aber eine Rasterzeile weiterschiebt, kann er diese Position nie erreichen. Das Endergebnis, das wir dadurch erhalten ist Folgendes:

1. Der FLD-Effekt greift, und wir lassen den VIC die nächste Charakterzeile eine Rasterzeile später zeichnen.
2. Der interne Adresszeiger des VIC auf die Daten der nächsten Charakterzeile wurde um 40 Zeichen (also eine Charakterzeile) erhöht.
3. * Da der VIC beim Erreichen der nächsten Rasterzeile wieder glaubt, er müsse sich auf die Charakterzeile vorbereiten, zählt er den internen Adresszeiger um weitere 40 Bytes nach oben und wartet auf die richtige Startposition zum Lesen der nächsten 40 Bytes.
4. Lassen wir ihn nun tatsächlich die Charakterzeile lesen, indem wir die FLD(VSP)-Routine beenden, so stellt er zwar, wie beim normalen FLD-Effekt, wieder Charakterzeilen dar, jedoch wurde durch das zweimalige Erhöhen des Adresszeigers um 40 Bytes eine gesamte Charakterzeile übersprungen! Hatten wir den FLD-Effekt also z.B. vor Erreichen der ersten Charakterzeile einsetzen lassen, so zeichnet der VIC nun nicht die erste, sondern die ZWEITE Charakterzeile, da er sich ja 40 Bytes zu weit nach vorne bewegte!!! Die erste Charakterzeile wurde somit ÜBERSPRUNGEN!

Im Klartext bedeutet das: für jede Rasterzeile, die wir die FLD(VSP)- Routine länger laufen lassen, überspringt der VIC eine Charakterzeile. Auf diese Weise können wir also auch die Daten, die in der Mitte des Video-RAMs liegen am Anfang des Bildschirms anzeigen lassen.
Der VIC liest nun aber pro Rasterdurchlauf immer 25 Charakterzeilen. Lassen wir ihn also durch unseren VSP-Trick 40 Bytes zu spät auf die vermeintliche 1.Charakterzeile zugreifen (es handelt sich um die 1 Charakterzeile die auf dem Bildschirm zu sehen ist, jedoch mit den Daten, die eigentlich erst in der zweiten Charakterzeile des Bildschirms erscheinen sollten), so hört für ihn der Bildschirm auch 40 Bytes zu spät auf.
Das Ergebnis: in der 25 . Bildschirmzeile liest und stellt der VIC die Daten des sonst ungenutzten Bereichs des Video-RAMs im Adressbereich von $07E8 bis $07FF dar (wenn

wir von der Standard-Basisadresse des Video-RAMs bei $0400 ausgehen). Dies sind 24 Zeichen, die somit am Anfang der letzten Bildschirmzeile zu sehen sind. Hieraufhin folgen 16 weitere Zeichen, die sich der VIC wieder aus der Anfangsadresse des Video-RAMs holt.
Zum besseren Verständnis sollten wir vielleicht noch klären, wie der VIC auf das Video-RAM zugreift. Selbiges ist immer 1024 Bytes (also exakt 1KB) lang, obwohl der Bildschirm nur 1000 Bytes groß ist und somit die letzten 24 Bytes nie sichtbar werden. Wie Sie nun wissen, kann der VIC immer nur 16KB der 64KB des C64 adressieren, wobei die Lage dieses Bereichs allerdings auch verschoben werden kann. Das heißt also, daß der VIC insgesamt zwar Adressen in einem 64KB-Raum (16 Adressbits sind hierzu notwendig) adressieren, aber gleichzeitig immer nur auf 16 KB für Grafikdaten zugreifen kann. Möchte man diesen 16KB-Bereich verschieben, so kann das mit den untersten zwei Bits von Adresse $DD00 getan werden. Im Normalfall stehen beide auf 0 und lassen den VIC deshalb im Bereich von $0000 bis $3FFF arbeiten. Diese Adressen dienen als Basis für den VIC die ihm von der NMI-CIA in die obersten zwei Adressbits für seine RAM-Zugriffe eingeblendet werden. Die Zugriffsadresse ist also 16 Bit groß, wobei die beiden Bits 0 und 1 aus $DD00 immer in den obersten zwei Bits (14 und 15) erscheinen. Die Bits 10-13 dieser Adresse bestimmen die Basisadresse des Video-RAMs.
Sie werden von den Bits 4-7 der VIC-Adresse $D018 geliefert. Steht hier der Bitcode $0001(so wie nach Einschalten des Computers), so wird also zusammen mit den Bits aus $DD00 die Adresse $0400 angesprochen, die im Normalfall die Basisadresse für das Video-RAM ist. Die restlichen, zur Adressierung benötigten 10 Bits kommen aus dem oben schon erwähnten, VIC internen Adresszähler. Er ist exakt 10 Bits lang, womit maximal 1 KB-Bereiche adressiert werden können.
Dadurch erklärt sich auch, warum der VIC durch austricksen mit der VSP-Routine, in der letzten Zeile die 24 sonst ungenutzten Zeichen des Video-RAMs darstellt: Da der 10-Bit-Zähler nun auf den Offset $03 E8 zeigt, werden von dort auch die Daten gelesen. Hierbei findet dann aber nach dem 24. Byte ein Überlauf des 10-Bit-Zählers statt, womit selbiger wieder auf 0 zurückspringt und wieder den Anfang des Video-RAMs adressiert.
Dadurch erklärt sich auch, warum in der letzten Bildschirmzeile nach den Zeichen aus $07E8-$07FF wieder die Zeichen ab Adresse $0400 erscheinen.
Bleibt noch zu erwähnen, daß dasselbe für das Color-RAM bei $ D800 gilt. Dies ist immer an einer Fixadresse, wobei die untersten 10 Bit ebenfalls aus dem Charakterzeilen-Adresszähler kommen. Das heißt also, daß die 24 sonst unsichtbaren Zeichen ihre Farbe aus dem ebenfalls sonst ungenutzten Color-RAM-Bereich von $DBE8-$DBFF erhalten.

4. Probleme die bei VSP entstehen können (im original ist dies Punkt 5)

Die Nachteile, die durch die Charakterverschiebung entstehen, sollten auch nicht unerwähnt bleiben:
Durch die Verschiebung der Video-RAM- Daten am Ende des Bildschirms um 24 Zeichen nach rechts, ist natürlich auch die gesamte Grafik in zwei Teile zerlegt worden, weswegen ein Scroller nicht ganz so einfach durchzuführen ist. Um diesen Fehler auszugleichen müssen wir einen zweiten Video-RAM-Bereich verwalten, in dem die gesamte Grafik um 24 Zeichen versetzt abgelegt sein muß. In diesem Video-RAM sind dann die ersten 24 Zeichen ungenutzt. Durch einen stinknormalen Rasterinterrupt können wir dann problemlos vor Beginn der Charakterzeile, an der der VIC-Adresszeiger überläuft, auf das zweite Video-RAM umschalten, wobei dieses dann zwar an der überlaufenden Adresse ausgelesen wird, was jedoch durch unsere Verschiebung wieder ausgeglichen wird.
Ein weiterer Nachteil, der bei VSP entsteht, liegt auf der Hand: Die letzten acht Bytes des jetzt sichtbaren Video-RAMs sind die Adressen, in denen die Spritepointer abgelegt werden. Das heißt also, daß wir beim gleichzeitigen Anzeigen von Grafikzeichen in diesen acht Bytes mit der Spriteanzeige Probleme bekommen. Jedoch auch dies ist zu

bewältigen. Entweder, indem man so geschickt arbeitet, daß in diesen Bereichen auf dem Bildschirm nur Zeichen in der Hintergrundfarbe zu sehen sind (bei schwarzem Hintergrund einfach das Color-RAM in den Adressen von $DBF0-$DBFF ebenfalls auf schwarz setzen). Oder aber indem wir in diesen Zeichen immer nur dann die eigentlichen Video-RAM- Werte eintragen, wenn sie ausgerechnet vom VIC benötigt werden (also auch exakt vor dem Lesen der Überlauf-Charakterzeile), und ansonsten die Spritepointer hineinschreiben. Da der VIC schon 42 Taktzyklen nach Beginn der Rasterzeile, in der die Charakterzeile beginnen soll, die Video-RAM-Daten gelesen hat, können wir also getrost wieder die Spritepointer zurückschreiben und haben so lediglich eine einzige Rasterzeile, in der die Sprites gar nicht (oder nur verstümmelt) dargestellt werden können.
Soll der gesamte Bildschirm einmal durchgescrollt werden können, so muß davon ausgegangen werden, daß maximal 25 Rasterzeilen am Beginn des Bildschirms ungenutzt bleiben müssen, da in Ihnen das Timing für den VSP-Effekt unterbracht werden muß. Da pro Rasterzeile der interne Adresspointer um eine Charakterzeile weitergezählt wird, muß also im Maximalfall in 25 Rasterzeilen der VSP-Effekt eingesetzt werden. Will man den Bildschirm nun konstant an eine bestimmten Position beginnen lassen, so muß nach der Anzahl der zu überspringenden Charakterzeilen wieder eine normale FLD-Routine (ohne das zusätzlich NOP) benutzt werden, um bis zur 25 . Rasterzeile nach Bildschirmanfang zu verzögern, OHNE daß gleich alle 25 Charakterzeilen übersprungen werden.

5. Weitere Programmbeispiele (im original ist dies Punkt 6)
Außer der oben schon erwähnten "VSP1"-Routine finden Sie auf dieser MD auch noch zwei weitere Beispielprogramme, mit den Namen "VSP2" und "VSP3"(wie alle unserer Beispiele werden auch sie mit "SYS4096" startet). In Ersterem haben wir zusätzlich zum VSP-Effekt die unteren drei Bits von $D011 zum Softscrollen des Bildschirms verwendet, so daß der Bildschirm weich nach oben und unten geschoben werden kann. Die Routine "VSP3" scrollt das gesamte Video-RAM ununterbrochen durch, wodurch quasi ein unendlicher Horizontalscroller durchgeführt werden kann. Die 25 unbenutzten Rasterzeilen am Bildschirmanfang, die für das VSP-Timing benötigt werden, sind mit Sprites hinterlegt, in denen man z.B. in einem Spiel auch ein Scoreboard unterbringen kann.
In der nächsten Folge des IRQ-Kurses werden wir uns den horizontalen Hardwarescroller "HSP" anschauen, bei dem das VSP-Prinzip auf die Horizontale Bildschirmposition umgesetzt wurde.

(ih/ub)

Teil 15 – Magic Disk 01/95

Im letzten Kursteil hatten wir uns einen ganz besonderen Rastertrick angeschaut. Anhand einer VSP-Routine hatten wir gelernt, wie einfach es ist, den Bildschirm des C64 HARDWAREMÄSSIG, also ohne den Prozessor mit großen Bildschirm-Verschiebe-Aktionen zu belasten, nach oben und unten zu scrollen. Dabei unterschied sich die VSP-Routine von einer FLD-Routine in nur einem NOP-Befehl, der das nötige Timing erzeugte, um den gewünschten Effekt auszulösen. Der FLD-Effekt war, wie wir gesehen hatten maßgeblich daran beteiligt, daß der VIC das Lesen von einigen Charakterzeilen vergaß, weswegen wir in der Lage waren, einzelne Bereiche im Video-RAM zu überspringen, und so den Bildschirm beliebig nach oben und unten zu scrollen. In diesem Kursteil soll es nun um einen nahen Verwandten von VSP gehen. Wir werden den HSP-Effekt besprechen."HSP" steht für "Horizontal Screen Position" und setzt den VSP-Effekt quasi auf die Horizontale um. Mit ihm können wir DEN GESAMTEN BILDSCHIRM problemlos um bis zu 320 Pixel nach rechts verschieben, wobei wir den Prozessor nur läppische 3 Rasterzeilen lang in Anspruch nehmen müssen. Damit werden schnell scrollende Baller

oder Jump'n Run Spiele, wie auf dem Amiga oder dem Super-NES von Nintendo auch auf dem C64 möglich!

1. Zum Prinzip von HSP

Wir erinnern uns: Um die VSP-Routine funktionsfähig zu machen, mussten wir den FLD-Effekt so anwenden, daß wir dem VIC vorgaukelten, sich auf das Lesen der nächsten Rasterzeile vorzubereiten und seinen internen Lesezähler hoch zuzählen, um die richtige Charakterzeilen-Adresse anzusprechen. Als er nun seinen Lesevorgang durchführen wollte machten wir ihn durch FLD-Wegdrücken der Charakterzeilen glauben, daß er doch noch nicht die entsprechende Zeile erreicht hatte. Somit übersprang er mehrere Adressen und somit auch Charakterzeilen. Wir ließen ihn also die Charakterzeilen zu spät lesen, was zu dem gewünschten Effekt führte.
Einen ähnlichen Trick können wir nun auch für die HSP-Routine anwenden. Wie wir ja wissen, so liest der VIC ab der Rasterposition $30 alle 8 Rasterzeilen die 40 Zeichen, die in den nächsten 8 Rasterzeilen zu sehen sein sollen, aus dem Video-RAM, um sie anschließend anzuzeigen. Durch den FLD-Effekt haben wir nun schon oft genug die erste Charakterzeile auf dem Bildschirm vor ihm hergeschoben, so daß der VIC diese Zeile verspätet erreichte, und somit der Bildschirm nach unten weggedrückt wurde. Der Witz ist, daß dieser Trick nun auch in her Horizontalen funktioniert! Denn sobald der VIC merkt, daß er sich in einer Charakterzeile befindet, in der er Zeichendaten zu Lesen und Anzuzeigen hat, beginnt er auch unverzüglich mit dieser Arbeit, und das ohne noch auf die horizontale Rasterposition zu achten, um ggf. festzustellen, daß er sich gar nicht am Zeilenanfang befindet! Wenn wir nun also eine Art FLD-Routine einsetzen, die nur für einen Teil der aktuellen Rasterzeile vorschreibt, daß selbige noch keine Charakterzeile ist, und dann mitten innerhalb dieser Zeile von uns wieder auf normale Darstellung zurückgeschaltet wird, so fängt der VIC auch prompt mittendrin damit an die Charakterdaten zu lesen und sofort auf den Bildschirm zu bringen. Verzögern wir also nach einem FLD bis zur Mitte der Rasterzeile, und schalten dann wieder zurück, so wird der Video-RAM Inhalt um exakt 20 Zeichen nach rechts versetzt auf dem Bildschirm dargestellt, wobei die 20 letzten Zeichen, die ja nicht mehr in diese Textzeile passen, automatisch erst in der nächsten Zeile erscheinen. Es kommt sogar noch besser: aufgrund eines internen Timers des VIC, der das Lesen der Charakterzeilen mitbeeinflusst (es sei den wir tricksen ihn aus) führt der VIC den Lesefehler in JEDER WEITEREN Charakterzeile ebenso durch, so daß es genügt, lediglich am Bildschirmanfang einmal um einen bestimmten Wert zu verzögern, um DEN GESAMTEN Bildschirm wie gewünscht nach rechts versetzt darzustellen!!!
Um den Trick nun umzusetzen müssen wir wie folgt vorgehen: zunächst schalten wir in $D011 den vertikalen Verschiebe-Offset (wird in den untersten 3 Bits festgelegt) auf 1, so daß für den VIC die erste Charakterzeile erst eine Rasterzeile nach Beginn des sichtbaren Bildschirmfensters folgt. Dieser Beginn liegt normalerweise in Rasterzeile $30.
Durch die Verschiebung legen wir die erste vom VIC zu lesende Charakterzeile jedoch in Rasterzeile $31. Verzögern wir nun jedoch in Rasterzeile $30 um eine bestimmte Anzahl Taktzyklen, und schalten wir dann die horizontale Verschiebung mittendrin wieder auf 0 zurück, so merkt der VIC plötzlich, daß er sich doch schon in einer Charakterzeile befindet und fängt eifrig damit an die Charakterdaten zu lesen und auf dem Bildschirm darzustellen. Wohlgemerkt obwohl er sich schon nicht mehr am Zeilenanfang befindet, sondern mitten innerhalb dieser Rasterzeile! Für jeden Taktzyklus, den wir mehr verzögern, stellt der VIC die Charakterdaten um jeweils ein Zeichen (also 8 Pixel) weiter rechts dar. Schalten wir also 10 Takte nach Beginn des linken Bildrandes die vertikale Verschiebung ab, so wird die Charakterzeile exakt 10 Zeichen nach rechts versetzt gezeichnet. Dies setzt sich, wie oben schon erwähnt, über den gesamten Bildschirm, also auch für die folgenden 24 weiteren Charakterzeilen, fort, womit auch der gesamte

Bildschirm um 10 Zeichen versetzt angezeigt wird!

2. Die Umsetzung

Wie das Ganze dann aussieht, können Sie sich in den Programmbeispielen "HSP1" und "HSP2" anschauen. Sie werden beide mit ",8,1" geladen und durch SYS4096 gestartet. HSP2 unterscheidet sich von HSP1 nur darin, daß das Zeichenwirrwarr, das durch den HSP-Effekt in der ersten Rasterzeile zu sehen ist, mit einem schwarzen Rasterbalken unsichtbar gemacht wurde.
Kommen wir nun also zu der Routine, die uns diesen Effekt erzeugt. Sie ist vom Aufbau eigentlich recht einfach.
Zunächst einmal befindet sich eine Initialisierungsroutine ab Adresse $1000, die wie die meisten unserer IRQ-Routinen das ROM abschaltet, um den IRQ-Vektor am Speicherende verwenden zu können, und anschließend einen Border-Raster-IRQ initialisiert, der uns zunächst einmal, wie immer, den unteren und oberen Bildschirmrand abschaltet. Sie steht ab Adresse $1200 im Speicher und ist gefolgt von einer Routine zum Auslesen des Joysticks, sowie der Routine "Control", die die Joystick-Daten auswertet, den Bildschirmverschiebeoffset berechnet und damit die HSP-IRQ- Routine beeinflusst. Selbige ist ab Adresse $1100 zu finden.
Dir Border-IRQ-Routine legt diese Adresse in den IRQ-Vektoren bei $FFFE/$FFFF ab, und gibt dem VIC vor, den nächsten Raster-IRQ in Rasterzeile $2D auszulösen.
Wird unsere HSP-Routine nun in dieser Rasterzeile aufgerufen, so glätten wir zunächst den IRQ, nach der uns mittlerweile altbekannten Methode. Ab dem Label "Onecycle" steht nun unsere eigentliche Routine, die wie folgt aussieht:

```
onecycle    LDA #$19        ;Bildsch.1 Zeile nach
            STA $D011       ;unten scrollen
            LDY #$08        ;Verzögerungsschleife
wy          DEY             ;um den richtigen
            BNE wy          ;Moment abzupassen
            JSR cycles      ;Verzögerung 12 Takte
            LDA #$18        ;Wert f. 1 Zeile zurück
redu1       BEQ redu2       ;2 oder 3 Take verzögern
redu2       BNE tt          ;ans Ende verzweigen
            NOP             ;20 NOPs die für das
            NOP             ;Timing später SEHR
            NOP             ;WICHTIG sind, ob-
            NOP             ;wohl sie hier
            NOP             ;übersprungen werden!!
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
tt          STA $D011       ;Wert schreiben
            LDA #$0E        ;Bildschirmfarben set
            STA $D020       ;zen ( innerhalb dieser
```

```
                LDA #$06        ;Befehle wird die Cha
                STA $D021       ;rakterz. Gelesen!)
```

Zu Beginn unserer IRQ-Routine wird zunächst also eine vertikale Verschiebung um eine Rasterzeile in $D011 eingetragen (Bits 0-2 enthalten den Wert %001=$01). Dadurch, daß der HSP-IRQ in Rasterzeile $2D aufgerufen wurde, und daß die IRQ-Glättung 2 Rasterzeilen verbraucht, befinden wir uns nun also in Rasterzeile $2F. Wir verzögern nun noch mit der folgenden Zählschleife und dem JSR-Befehl um eine knappe weitere Zeile, wobei exakt die Position abgepasst wird, an der sich der Rasterstrahl genau am Anfang des linken Randes des sichtbaren Bildschirms befindet (so wie bei der Routine zum Abschalten der seitlichen Bildschirmränder). Hierbei gehört der "LDA #$18"- Befehl ebenfalls zur Verzögerung. Er initialisiert den Akku jedoch gleichzeitig schon mit dem Wert vor, den wir in $D011 schreiben müssen, um die vertikale Verschiebung wieder abzuschalten (die untersten drei Bits enthalten den Wert 0) . Nun folgt wieder eine sehr ausgeklügelte Verzögerung, um Taktgenau eine ganz bestimmte Rasterposition abzupassen. Beachten Sie bitte, daß die Routine an dieser Stelle von der oben schon erwähnten "Control"- Routine modifiziert wird, um die jeweils gewünschte Bildschirmverschiebung zu erreichen. Wir haben hier nun zwei Branch-Befehle, die jeweils mit Labels versehen sind, und denen 20 NOP-Befehle folgen. Zum Schluß steht dann der STA-Befehl, mit dem der Wert des Akkus in $D011 eingetragen wird, um nach der gewünschten Verzögerung den VIC zum Lesen der Charakterzeile zu bewegen. Schauen wir uns nun zunächst die beiden Branches und deren Bedeutung an:

```
                lda #$18        ;Wert f.1 Zeile zurück
redu1           beq redu2       ;2 oder 3 Take verzögern
redu2           bne tt          ;ans Ende verzweigen
```

Durch den vorherigen LDA-Befehl, der einen Wert ungleich Null lädt, wissen wir, daß das Zero-Flag in jedem Fall gelöscht ist. Außerdem scheint der "BEQ REDU2"-Befehl unsinnig zu sein, zumal er exakt zum nächsten Befehl weiterspringt.
Wie aber auch schon bei unserer IRQ-Glättungsroutine hat dieser Branch-Befehl die Aufgabe, exakt einen Taktzyklus lang zu verzögern. Ein Branch-Befehl benötigt mindestens zwei Takte um ausgeführt zu werden. Trifft die abgefragte Bedingung nicht zu, so wird gleich beim nächsten Befehl fortgefahren (so wie das hier auch der Fall ist). Trifft die Bedingung jedoch zu, so dauert der Branch-Befehl einen Taktzyklus länger, in dem der Prozessor den Sprung-Offset auf den Programmzähler aufaddieren muß. Soll nun um eine gerade Anzahl Zyklen verzögert werden, weil der Bildschirm um eine gerade Anzahl Zeichen verschoben werden soll, so steht hier ein BEQ-Befehl, der nur 2 Zyklen verbraucht. Soll eine ungerade Anzahl verzögert werden, so wird von der Routine " Control", im Label "REDU1" der Opcode für einen BNE-Befehl eingetragen, womit die Routine einen Taktzyklus länger dauert, und somit auch ungerade verzögert. Der nun folgende BNE-Befehl ist immer wahr und verzögert somit immer 3 Taktzyklen (da dies eine ungerade Zahl ist, verhält es sich also eigentlich umgekehrt mit der Änderung des BEQ-Befehls für gerade und ungerade Verzögerung). Bei diesem Befehl wird von "Control" die Sprungadresse modifiziert. Je nach dem, wieviele weitere Zyklen verzögert werden müssen, trägt die Routine einen Offset auf die folgenden NOP-Befehle ein. Der Befehl verzweigt dann nicht mehr auf das Label "TT", wo der Akkuinhalt nach $D011 geschrieben wird, sondern auf einen NOP-Befehl davor. Einer dieser Befehle verzögert dann immer um 2 Taktzyklen. Hierzu sollten wir einen Blick auf die Routine "Contol" werfen:

```
control         LDX #$D0        ;Opcode für BNE in
                STX redu1       ;REDU1 ablegen
                LDA xposhi      ;X-Verschiebungs-
```

```
                STA xposhib         ;zähler kopieren
                LDA xposlo          ;(Low- und High-
                STA xposlob         ;byte)
                AND #$08            ;Bit 3 v. XLo ausmask.
                BNE co1             ;<>0, dann
                LDX #$F0            ;Opcode für BEQ in
                STX redu1           ;REDU1 ablegen
co1             LSR xposhib         ;X-Verschiebung
                ROR xposlob         ;(16 Bit) durch 4X
                LSR xposhib         ;Rotieren nach rechts
                ROR xposlob         ;mit 16 dividieren
                LSR xposhib
                ROR xposlob
                LSR xposhib
                ROR xposlob
                SEC                 ;Subtraktion vorbereiten
                LDA #$14            ;20 NOPS
                SBC xposlob         ;minus XPos/16
                STA redu2+1         ;als Sprungoffset für BNE bei REDU2
                LDA xposlo          ; Alte X-Versch. Laden
                AND #$07            ;unterste 3 Bits isolieren
                ORA #%00011000      ;Standard-Bits setzen
                STA softmove+1      ;in hor. Scroll eintragen
                RTS
```

Zuallererst trägt die Control-Routine, die innerhalb des Border-IRQs aufgerufen wird, in das Label "REDU1" den Wert $D0 ein, der dem Opcode für den BNE-Befehl entspricht. Hieran anschließend werden die Inhalte der Labels "XPOSLO" und "X-POSHI" in die Labels "XPOSLOB" und "X-POSHIB" kopiert, was für die Offset-Berechnung später notwendig ist. Diese Labels sind im Source-Code des Programms auf die Zeropage-Adressen $02,$03,$04 und $05 vordefiniert."XPOSLO" und "X-POSHI" enthalten einen 16- Bit Zähler für die horizontale Bildschirmverschiebung, die von der Joystickabfrage, die ebenfalls während des Border-IRQs aufgerufen wird, den Joystickbewegungen entsprechend hoch oder runter gezählt wird.
Dieser Zähler kann also einen Wert zwischen 0 und 320 enthalten. Da mit der HSP-Routine der Bildschirm nur in Schritten von einzelnen Zeichen (also 8 Pixeln) versetzt werden kann, muß unsere Routine zum weichen Scollen des Bildschirms auch noch den horizontalen Verschiebeoffset des Bildschirms verändern.
Diese Verschiebung wird in Register$D016 des VICs festgelegt, wobei die untersten 3 Bits unseres XPOS-Wertes in die untersten 3 Bits dieses Registers gelangen müssen. Die Bits 3-9 des XPOS-Wertes enthalten nun (dreimal nach rechts verschoben) die Anzahl Zeichen, und somit auch Taktzyklen, die die HSP-Routine verzögern muß, damit der VIC die Charakterzeile erst an der gewünschten Position liest. Ist also das 3 . Bit (das das 0. Bit der Anzahl ist) gesetzt, so muß eine ungerade Anzahl Zeichen verzögert werden. In dem Fall enthält das Label "REDU1" schon den richtigen Wert, nämlich den Opcode für den BNE-Befehl, der zusammen mit dem BNE-Befehl bei "REDU2" sechs (also eine gerade Anzahl) Taktzyklen verbraucht. Ist Bit 3 gelöscht, so enthält der Akku nach dem " AND #$08"-Befehl den Wert 0 und es wird vor dem Weitergehen im Programm der Opcode für den BEQ-Befehl in "REDU1" eingetragen. Damit wird in der HSP-Routine um 2+3=5(also eine ungerade Anzahl) Taktzyklen verzögert. Nun muß noch ermittelt werden, wieviele zusätzliche NOPs zur Verzögerung notwendig sind. Da ein NOP-Befehl immer 2 Taktzyklen braucht, wird nur die halbe Anzahl NOPs benötigt, um entsprechend viele Zeichen (Taktzyklen) lang zu verzögern. Da zudem noch die 3 Bit Verschiebeoffset in XPOS stehen, muß dieser Wert durch 16 dividiert werden, um den gewünschten Wert zu erhalten. Diese Berechnung wird an der Kopie von XPOS, also in den Labels "X-POSLOB"

und "XPOSHIB" durchgeführt.
Nachdem im Low-Byte dieser Register hiernach die Anzahl der benötigten NOPs stehen, kann nun die Sprungadresse für den BNE-Befehl bei "REDU2" berechnet werden. Da Branch-Befehle immer nur ein Byte mit dem relativen Offset zum aktuellen Programmzähler enthalten, müssen wir also lediglich angeben, wieviele NOPs übersprungen werden müssen. Dies wird durch die Gesamtanzahl NOPs minus der benötigten Anzahl NOPs errechnet, und in Adresse " REDU2"+1 eingetragen.
Die Verzögerung sollte nun also sauber funktionieren.
Zu guter Letzt wird noch der horizontale Verschiebeoffset für das horizontale Softscrolling ermittelt, indem aus "XPOSLO" die untersten drei Bits ausmaskiert werden. Da diese Bits im Register $D016 landen müssen, das auch für die 38/40-Zeichendarstellung und den Multicolormodus zuständig ist, setzen wir die entsprechenden Bits mit dem folgenden OR-Befehl. Der resultierende Wert wird in das Label "SOFTMOVE"+1 eingetragen, das in der HSP-Routine kurz vor dem oben gezeigten Codestück steht:

```
softmove    LDA #$00
            STA $D016
```

Hier wird also lediglich der Verschiebeoffset in den Operanden des LDA-Befehls eingetragen, der dann in der HSP-Routine die Bildschirmverschiebung jeweils wie benötigt in $D016 einträgt.
Damit hätten wir alle Einzelheiten der HSP-Routine besprochen. Wie Sie sehen funktioniert sie sogar noch einfacher als die VSP-Routine. Vielleicht experimentieren Sie einmal ein wenig mit den Programmbeispielen und versuchen einen horizontalen Endlosscroller daraus zu machen. Der HSP-Effekt funktioniert übrigens genauso wie VSP auch mit HIRES-Grafik. Im nächsten Kursteil werden wir auch das noch sehen, und die erstaunlichste IRQ-Raster- Routine kennenlernen die je entdeckt wurde: die AGSP-Routine nämlich, die eine Kombination aus HSP und VSP darstellt, und mit der es problemlos möglich ist den kompletten Bildschirm in ALLE Richtungen zu scrollen, ohne große Kopieraktionen mit dem Prozessor durchführen zu müssen!!!

(ih/ub)

Teil 16 – Magic Disk 02/95

Herzlich Willkommen zum 16. Teil unseres Kurses über die Rasterstrahlprogrammierung. In den letzten beiden Kursteilen hatten wir uns mit zwei IRQ-Routinen beschäftigt, die das hardwaremäßige Scrollen des Bildschirms jeweils in der Horizontalen (HSP-Routine) und Vertikalen (VSP-Routine) ermöglichten. Der Clou an diesen beiden Routinen war, daß des GESAMTE Bildschirm vom VIC verschoben wurde, und wir keinen einzigen Taktzyklus zum Kopieren von Bildschirmdaten verschwenden mussten. In diesem Kursteil wollen wir nun das "Sahnestückchen" der Raster-IRQ-Programmierung besprechen:
die Kombination aus HSPund VSP-Routine, genannt "AGSP". Diese Abkürzung steht für "Any Given Screen Position" und bedeutet, daß der Bildschirm in jede Richtung verschoben werden kann, und das natürlich HARDWAREMÄSSIG, also wie schon bei HSP und VSP ohne auch nur einen Zyklus zum Kopieren von Bildschirmdaten zu verschwenden!
Stellen Sie sich vor, was für Möglichkeiten sich hierdurch für Spiele ergeben: so kann man z.B. problemlos Spiele programmieren, die sich über einer beliebig großen Oberfläche abspielen, die in alle Richtungen flüssig gescrollt werden kann, ohne dabei mehr als 30 Rasterzeilen pro Frame zu verbrauchen. 25 Rasterzeilen lang ist der Prozessor eigentlich nur mit dem Abpassen der optimalen Rasterstrahlpostion für den HSP-Effekt beschäftigt,

der, wie wir schon im 14. Kursteil sehen konnten, pro Charakterzeile, die der Bildschirm horizontal gescrollt werden soll, eine Rasterzeile für das Timing verbraucht.
Damit Sie sehen, was mit der AGSP-Routine alles möglich ist, haben wir Ihnen natürlich wieder zwei Beispielprogramme vorbereitet. Sie heißen "AGSP1" und "AGSP2" und werden wie immer mit ",8,1" geladen und durch ein "SYS4096" gestartet. Des weiteren finden Sie auf dieser MD eine Demo mit Namen "AGSP-Demo", in der am eindrucksvollsten die Möglichkeiten der AGSP-Routine demonstriert werden. Sie sehen hier einige Hiresgrafiken, die in Sinuswellenform auf und abschwingen und gleichzeitig von links nach rechts scrollen. Hierbei werden jeweils neue Grafiken in die Grafik einkopiert, so daß der Eindruck entsteht, daß der Bildschirm weitaus größer sei, als eigentlich sichtbar. Das Demo laden Sie mit LOAD"AGSP-DEMO",8 und starten es durch "RUN" .

1. Das AGSP-Prinzip

Kommen wir nun also zum Arbeitsprinzip von AGSP, das eigentlich recht schnell erläutert ist: Zunächst einmal passen wir die erste Rasterzeile des Bildschirms ab, an der die erste Charakterzeile gelesen werden muß. Hier nun lassen wir eine VSP-Routine, so wie wir Sie im letzten Kurteil schon kennenlernten, zunächst dem VIC vorgaukeln, daß er auf den Beginn der Charakterzeile noch zu warten hat, indem wir eine horizontale Bildschirmverschiebung um 1 Pixel in $D011 angeben. Inmitten dieser Zeit soll nun die VSP-Routine diese Verschiebung wieder Aufheben, um so den VIC sofort die Charakterdaten lesen zu lassen, und ihn somit dazu zu zwingen, den Bildschirm nach rechts versetzt darzustellen. Die Anzahl Taktzyklen, die bis zum Zurückschalten verzögert werden müssen, muß dabei der Anzahl Zeichen entsprechen, die der Bildschirm verschoben werden soll. Hieran anschließend lasen wir eine HSP-Routine folgen, die den Beginn der nächsten Charakterzeilen vor dem VIC herdrückt, um so auch die zeilenweise Verschiebung des Bildschirmanfangs zu erreichen. Da diese Routine maximal 25 Rasterzeilen benötigt (nämlich dann, wenn die Darstellung erst 25 Charakterzeilen später beginnen soll, müssen die ersten 25 Rasterzeilen des sichtbaren Bildschirms immer für das Timing der AGSP-Routine verwendet werden, auch wenn stellenweise weniger Rasterzeilen dazu benötigt werden (z.B. wenn nur eine Zeile Verschiebung erzeugt werden muß) . Für diese Fälle müssen wir mit Hilfe einer normalen FLD-Routine den Bildschirm um die fehlende Anzahl Rasterzeilen nach unten drücken, damit der Bildschirm auch immer in ein und derselben Rasterzeile beginnt. Diese FLD-Routine wird in unserem Programmbeispiel den VSPund HSP-Routinen vorgeschaltet. Sie sorgt also gleichzeitig für immer gleiches Timing und zudem für die richtige Positionierung in der Horizontalen. Aus diesem Grund sind dann die ersten 25 Rasterzeilen des Bildschirms auch nicht zum Darstellen von Bildschirmdaten verwendbar.
In unserem Programmbeispielen haben wir hier jedoch Sprites untergebracht, in denen die Scrollrichtung angezeigt wird.
In einem Spiel könnte sich hier z. B. auch eine Score-Anzeige befinden.

2. Das Programm

Wollen wir uns nun die Funktionsweise von AGSP anhand des Programmbeispiels "AGSP1" anschauen. Die Routinen zur Joystickabfrage und Scrollgeschwindigkeitsverzögerung sollen hier nicht Thema unseres Kurses sein, sondern lediglich die Routine, die uns den Bildschirm beliebig im sichtbaren Fenster positioniert. Zur angesprochenen Scrollgeschwindigkeitsverzögerung sei erwähnt, daß wir hier beim Drücken und Loslassen des Joysticks die Scrollrichtung verzögern, bzw. nachlaufen lassen, so daß der Scrolleffekt zunächst träge anfängt und beschleunigt und beim Loslassen erst abgebremst werden muß, bis er zum Stillstand kommt.
Die Initialisierungsroutine zu AGSP1 bei $1000 legt nun zunächst einen Border-IRQ in Zeile $FB fest, wobei, wie schon so oft in unseren Beispielen, der Hard-IRQ-Vektor bei

$FFFE/$ FFFF benutzt wird.
Diese Border-Routine (bei $1400) schaltet nun den oberen und unteren Bildschirmrand ab und initialisiert einen Raster-IRQ für Rasterzeile $18, wobei dann unsere eigentliche IRQ-Routine "AGSP" angesprungen wird. Sie befindet sich an Adresse $1200 und soll im nun Folgenden ausführlich beschrieben werden. Die AGSP-Routine soll nun der Reihe nach folgende Arbeiten verrichten:

- Verwaltung des Sprite-Scoreboards, das aus zwei Spritereihen zu je acht Sprites bestehen soll.
- Glätten des IRQs, um ein möglichst exaktes Timing zu bewirken.
- FLD-Routine benutzen, um die fehlenden Rasterzeilen, die von VSP ggf. nicht benötigt werden, auszugleichen
- VSP-Routine Charakterweisen zum hoch- und runterscrollen des Bildschirms einsetzen
- HSP-Routine zum Charakterweisen links und rechtsscrollen des Bildschirms einsetzen
- horizontales und vertikales Softscrolling zur exakten Bildschirmverschiebung durchführen

2a. Die Spriteliste
Kommen wir nun also zum ersten Teil der AGSP-Routine, die zunächst das Sprite-Scoreboard verwaltet. Wir verwenden hier zwei Sätze zu je acht Sprites, deren Spritedaten in den Adressen von $0C00-$1000 zu finden sein sollen (Sprite-Pointer $30-$3f). Um die Pointer zwischen beiden leichter umschalten zu können benutzen wir zwei Video-RAM-Adressen. Für die erste Zeile, die eh in einem unsichtbaren Bildbereich liegt, wird das Video-RAM an Adresse $0000 verschoben, so daß wir die Spritepointer in den Adressen $03F8-$03FF liegen haben.
Für die zweite Spritezeile wird das Video- RAM an die (normale) Adresse $0400 gelegt, womit wir die Spritepointer an den Adressen $07F8-$07FF vorfinden. Dadurch, daß durch den VSP-Effekt auch diese Spritepointer als einzelne Zeichen auf dem Bildschirm auftauchen können, müssen wir die Pointer in jedem Rasterdurchlauf (Frame) wieder neu setzen, da die Pointer auf dem Bildschirm ja durchaus auch Bildschirmzeichen enthalten können. Doch kommen wir nun endlich zum Source-Code der AGSP-Routine, der in seinem ersten Teil zunächst einmal die Spriteinitialisierungen vornimmt und eigentlich einfach zu verstehen sein sollte:

```
agsp    PHA             ;IRQ-Beginn -> also
        TXA             ;Prozessorregister
        PHA             ;retten
        TYA
        PHA
        LDA #$1E        ;Y-Position aller
        STA $D001       ;Sprites auf Raster-
        STA $D003       ;zeile $1E (dez. 30)
        STA $D005       ;setzen
        STA $D007
        STA $D009
        STA $D00B
        STA $D00D
        STA $D00F
        LDA #$01        ;Die Farbe aller
        STA $D027       ; Sprites auf 1, also
        STA $D028       ;'weiß' setzen
        STA $D029
```

```
STA $D02A
STA $D02B
STA $D02C
STA $D02D
STA $D02E
LDA #$FF        ;Alle Sprites ein
STA $D015       ;schalten
LDA #$80        ;Hi-Bit d. X-Pos. Von
STA $D010       ;Sprite 8 setzen
LDA #$00        ;Sprite-Multicolor
STA $D01C       ;ausschalten
CLC             ;C-Bit f.Add. löschen
LDA #$58        ;XPos Sprite 0
STA $D000       ;setzen
ADC #$18        ;$18 (dez.24) addieren
STA $D002       ;u. XPos Spr.1 setzen
ADC  #$18       ;$18 (dez.24) addieren
STA $D004       ;und XPos Spr.2 setzen
ADC  #$18       ;usw.
STA $D006
ADC  #$18
STA $D008
ADC  #$18
STA $D00A
ADC  #$18
STA $D00C
ADC  #$18       ;$18 (dez.24) addieren
STA $D00E       ;und XPos Spr7 setzen
```

Bis hierhin hätten wir nun die wichtigsten Spritedaten initialisiert. Dadurch, daß diese in jedem Frame neu gesetzt werden, können Sie im AGSP-Bereich problemlos auch noch acht weitere Sprites sich bewegen lassen. Sie müssen deren Spritedaten dann nur am Ende der AGSP-Routine wieder in die VIC-Register einkopieren. Nun folgt noch die Initialisierung der Spritepointer. Zunächst die für die Sprites der ersten Spritezeile, deren Pointer ja im Video-RAM ab Adresse $0000 untergebracht sind, und sich somit in den Adressen von $03F8-$03FF befinden. Anschließend werden die Spritepointer für die zweite Spritereihe eingetragen, die sich im normalen Bildschirmspeicher bei $0400, in den Adressen $07F8-$07FF befinden. Das abschließende schreiben des Wertes 3 in Register $DD00 stellt sicher, daß der 16K-Adressbereich des VICs auf den unteren Bereich, von $0000 bis $3FFF, gelegt ist:

```
LDX #$30        ;Spr.Pointer Spr0=$30
STX $03F8       ;bis Spr7=$37 in
INX             ;die Pointeradressen
STX $03F9       ;$03F8-$03FF ein-
INX             ;tragen
STX $03FA
INX
STX $03FB
INX
STX $03FC
INX
STX $03FD
INX
STX $03FE
INX
STX $03FF
INX             ;Spr.Pointer Spr0=$38
```

```
                STX $07F8       ;bis Spr8=$3F in
                INX              die Pointeradressen
                STX $07F9       ;$07F8-$07FF ein-
                INX             ;tragen
                STX $07FA
                INX
                STX $07FB
                INX
                STX $07FC
                INX
                STX $07FD
                INX
                STX $07FE
                INX
                STX $07FF
                LDA #$03        ;VIC-Adressraum auf
                STA $DD00       ;$0000-$3FFF schalten
```

2b. Das Glätten des IRQs

Der nun folgende Teil der AGSP-Routine ist für das Glätten des IRQs verantwortlich. Außerdem wird hier der Border-IRQ wieder als nächste IRQ-Quelle festgelegt, womit diese Arbeit nun auch erledigt wäre:

```
                DEC $D019       ;IRQ-ICR freigeben
                LDA #$1D        ;Nächster Raster-IRQ
                STA $D012       ;bei Zeile $1D
                LDA #<irq2      ;Adresse Glättungs-
                STA $FFFE       ;IRQ in $FFFE/$FFFF
                LDA #>irq2      ;eintragen
                STA $FFFF
                CLI             ;IRQs erlauben
ch              NOP             ;Insgesamt 23 NOPs, in
                ...             ;denen der Glättungs-
                NOP             ; IRQ ausgelöst wird
                BNE ch
```

Beachten Sie bitte, daß das Programm hier nicht mehr normal fortgesetzt wird, sondern daß innerhalb der 23 NOPs auf den Glättungs-IRQ bei "IRQ2" gewartet wird. Selbiger folgt gleich diesem Stückchen Source-Code, wobei ihm ein RTS-Befehl vorangestellt ist, der mit dem Label "Cycles" assoziiert wird. Dieser RTS-Befehl wird später im AGSP-IRQ des öfteren zum Verzögern angesprungen.
Zusammen mit dem aufrufenden JSR-Befehl werden so 12 Taktzyklen verbraucht (je Befehl 6 Zyklen):

```
cycles:         RTS             ;Verzög.- Unterroutine
irq2            PLA             ;(IRQ-) Prozessorstatus
                PLA             ;und Rücksprungadresse
                PLA             ;vom Stapel werfen
                DEC $D019       ;VIC-ICR freigeben
                LDA #$F8        ;Nächster Raster-IRQ
                STA $D012       ;bei Zeile $F8
                LDA #<border    ;Adresse der Border-IRQ
                STA $FFFE       ;-Routine in Vektor
                LDA #>border    ;bei $FFFE/$FFFF ein-
                STA $FFFF       ;tragen
                LDA #$C8        ;40-Zeichen-Darstellung
                STA $D016       ; einschalten
                LDA #$06        ;Video-RAM nach $0000
                STA $D018       ;verschieben (für Sprite-Pointer 1.Sprite-Zeile, siehe
```

```
                        oberste 4 Bits!)
            NOP         ;verzögern
            LDA $D012   ;Und den letzten Takt
            CMP #$1D    ;Zyklus korrigieren
            BEQ line    ; IRQ-Glättung)
line        LDA #$00
            …
```

2c. Die FLD-Routine

Nun folgt der wichtige Teil der AGSP-Routine. Zunächst einmal noch ein wenig Verwaltungsarbeit. Wir setzen hier den Bildschirmbereich, in dem das VSP-Timing untergebracht ist auf schwarz und setzen schon einmal die Y-Positionen der Sprites für die zweite Spritezeile des Scoreboards (wie Sie aus unseren Kursteilen über die Spriteprogrammierung wissen, so wird die neue Y-Koordinate erst dann vom VIC übernommen, wenn die ersten Sprites fertig gezeichnet sind - somit können wir die neuen Koordinaten irgendwann setzen, wenn der VIC die alten Sprites noch zeichnet - was hier der Fall ist):

```
line        LDA #$00        ;Bildschirm und Rah
            STA $D020       ;mein auf ' schwarz'
            STA $D021       ;schalten
            JSR cycles      ;3*12=36 Zyklen
            JSR cycles      ; verzögern
            JSR cycles
            BIT $EA         ;3 Zyklen verzögern
            LDY #$1E+21     ;YPos=YPos+21
            STY $D001       ;und für die Spr0-
            STY $D003       ; Spr7 setzen
            STY $D005
            STY $D007
            STY $D009
            STY $D00B
            STY $D00D
            STY $D00F
```

Nach diesen mehr organisatorisch wichtigen Aufgaben folgt nun die FLD-Routine, die das Timing zur VSP-Routine ausgleichen soll:

```
fld         LDX #$27        ;Max. Anz. FLD-Durchl.
            LDY #$01        ;Index D011/ D018- Tab.
fldlp       JSR cycles      ;12 Zyklen verzögern
            LDA field1,Y    ;Wert für $d011 holen
            STA $D011       ;und setzen
            LDA field2,Y    ;Wert für $D018 holen
            STA $D018       ;und setzen
            NOP             ;6 Zyklen verzögern
            NOP
            NOP
            INY             ;Tab-Index+1
            DEX             ;FLD-Durchläufe-1
            CPX <fldcnt     ;=erforderliche Anz.
            BNE fldlp       ;Durchl.?Nein->Weiter
            NOP             ;Sonst 14 Zyklen
            JSR cycles      ;verzögern
            LDA field2,Y    ;und letzten Wert für
            INY             ;$d018 setzen
            STA $D018
```

Die FLD-Schleife benötigt für einen Durchlauf exakt eine Rasterzeile. In jedem Durchlauf wird vor Beginn der Charakterzeile durch die horizontale Verschiebung in $D011 der Charakterzeilenanfang vor dem Rasterstrahl hergedrückt. Dies bewältigen wir mit Hilfe einer Liste namens "Field1" . Sie wurde im Hauptprogramm der AGSP-Routine vorinitialisiert und befindet sich an Adresse $0100. Es stehen dort Werte für $D011, die die zur normalen Bildschirmdarstellung notwendigen Bits gesetzt haben, und deren Bits für der horizontale Bitverschiebung jeweils um eine Rasterzeile erhöht werden, und bei acht Rasterzeilen Verschiebung wieder auf 0 zurückgesetzt sind. Die Tabelle ist 64 Einträge lang und enthält somit für jede Rasterzeile ab dem Beginn der FLD-Routine einen passenden Wert für die$D011- Verschiebung. Sie wird ebenso in der VSP-Routine verwendet. Die Tabelle " Field2" erfüllt einen ähnlichen Zweck:
Sie enthält Werte für Register $D018, befindet sich an Adresse $0140 und enthält ebenfalls 64 Werte. Auch ihre Werte gelangen in jeder Rasterzeile (also auch jedem FLD-Durchlauf) in das Register $D018, womit wir den Adressbereich des Video-RAMs verschieben. Die Tabelle enthält 20 Einträge, die das Video-RAM an Adresse $0000 legen, gefolgt von 44 Einträgen, die es an Adresse $0400 verschieben. Auf diese Weise schaltet Sie also exakt in der 21. Rasterzeile nach Beginn der FLD-Routine auf den Bildschirm bei $0400 um, womit wir die Spritepointer auch auf diesen Bereich umschalten. Nach dieser 21. Rasterzeile hat der VIC nämlich gerade die erste Spritereihe fertig gezeichnet und wir bringen ihn so also dazu die auch noch die zweite Reihe mit anderen Pointern zu zeichnen. Der Grund, warum dies über eine Tabelle geschehen muß, und nicht etwa durch Abpassen der entsprechenden Position und dem dann folgenden Umschalten liegt auf der Hand: Braucht die der FLD-Routine folgende VSP-Routine z.B. 25 Rasterzeilen, um den Bildschirm 25 Charakterzeilen tiefer darzustellen, so läuft unsere FLD-Routine nur einmal durch und endet, wenn die Sprites noch längst nicht ganz fertig gezeichnet sind. Umgekehrt kann es auch passieren, daß die VSP-Routine keine Zeit benötigt, weil keine Verschiebung notwendig ist, und deshalb die FLD-Routine 25 Rasterzeilen lang laufen muß, damit der Bildschirm an derselben Position wie im letzten Frame erscheint. In dem Fall muß das Umschalten von der FLD-Routine durchgeführt werden. Benutzen allerdings beide Routinen ein und dieselbe Tabelle und denselben Index darauf, so übernimmt automatisch die Routine die Umschaltungsaufgabe, die gerade an der Reihe ist, ohne, daß wir etwas dazutun müssen! Dies mag verwirrender klingen als es ist: Im Endeffekt stellt die Tabelle sicher, daß immer in der 21. Rasterzeile seit FLD-Beginn, die Spritepointer umgeschaltet werden - unabhängig davon, ob sich der Prozessor zu diesem Zeitpunkt noch in der FLD oder schon in der VSP-Routine befindet!

2d. Die VSP-Routine

Nach FLD folgt die VSP-Routine, die sich von der ersteren nur darin unterscheidet, daß sie mit zwei Zyklen Verzögerung die Änderungen in $D011 einträgt und somit nicht nur den Beginn der nächsten Charakterzeile, sondern auch den VIC internen Adresszeiger selbiger erhöht und somit eine Charakterzeile überspringt:

```
vsp     INX             ;Alter FLD-Zähler=
        STX <vspcnt     ;VSP-Zähler+1
        NOP             ;8 Takte verzögern
        NOP
        NOP
        NOP
vsplp   NOP             ;8Takte verzögern
        NOP
        NOP
        NOP
```

```
LDX field1+3,Y      ;Wert aus d011-Tab+3
STX $D011           ;nach $D011 kopieren
NOP                 ;Nochmals 6 Takte
NOP                 ; verz. (Ausgleich
NOP                 ; zu FLD)
LDA field2,Y        ;Wert aus D018-Tab
STA $D018           ;nach $D018 kopieren
NOP                 ;4 Zyklen bis Ende
NOP                 ; verzögern
INY                 ;Tab-Index+1
DEC <vspcnt         ;VSP-Zähler-1
BNE vsplp           ;<>0 -> weiter
BIT $EA             ;Sonst 7 Takte
NOP                 ;verzögern
NOP
```

Wie Sie sehen, ist dies quasi unsere FLD-Routine. Einziger Unterschied liegt in der Art, wie die beiden Tabellen ausgelesen und geschrieben werden. Um den VSP-Effekt zu erzielen kann dies hier nicht mehr direkt aufeinanderfolgen.
Außerdem wird hier nicht mehr der nächste Tabellenwert von "Field1" gelesen, sondern der dritte Wert danach. Dies tun wir, um in jedem Fall eine $D011-Wert zu schreiben, der die Charakterzeile mindestens 1 Rasterzeile vor den Rasterstrahl drückt. Durch die Zeit die zwischen FLD und VSP vergeht haben wir nämlich auch schon eine Charakterzeile verloren, und damit der Y-Index nicht unnötig erhöht werden muß greifen wir einfach auf einen weiter entfernten Tabellenwert zu (wie viele Rasterzeilen wir die Charakterzeile vor uns herschieben ist ja egal - es zählt nur, daß sie vor uns hergeschoben wird)! Die Anzahl der VSP-Schleifendurchläufe wird durch den FLD-Zähler ermittelt. In der FLD-Routine wurde das X-Register mit dem Wert $27 initialisiert. Nach Abzug der FLD- Durchläufe enthält das X-Register nun noch die erforderliche Anzahl VSP-Durchläufe, die in dem Label "VSPCNT" (Zeropageadresse $069) bgelegt wird und von nun an als Zähler dient.

2e. Die HSP-Routine

Nun folgt dem ganzen noch die HSP-Routine, die Sie ja noch aus dem letzten Kursteil kennen. Wir schreiben hier zunächst den nächsten Wert der Field1- Tabelle in $D011 und verzögern dann bis zum gewünschten Punkt um den geschriebenen $D011-Wert-1 zu schreiben, mit dem die horizontale Bildverschiebung erzielt wird:

```
HSP     LDX field1+3,Y      ;nächsten $D011-Wert
        STX $D011           ;schreiben
        JSR cycles          ;Anfang sichtbaren Bildschirm abwarten
        DEX                 ;$D011-Wert-1
redu1   BEQ redu2           ;Ausgleich für unge-
redu2   BNE tt              ;rade Zyklen
        NOP                 ;Insgesamt 20
        ...                 ;NOPs für das
        NOP                 ; HSP-Timing
tt      STX $D011           ;akt.Z. einschalt.
```

Hier also drücken wir zunächst den Charakterzeilenbeginn vor den Rasterstrahl und warten bis zum benötigten Zeitpunkt um die vertikale Bildschirmverschiebung um eine Rasterzeile herunterzustellen und so den HSP-Effekt zu erzielen. Die merkwürdige BEQ/BNE-Folge dient dem Ausgleichen eines ggf. ungeraden Verzögerungszeitraums. Das Label "tt" wird von der Timingroutine verändert um so verschiedene Zeitverzögerungen innerhalb der 20 NOPs zu erreichen.

2f. Das Softscrolling
Dies ist die leichteste Aufgabe unserer IRQ-Routine. Es werden hier lediglich die Softscrollingwerte in der Horizontalen und Vertikalen in die Register $D016 und $D011 eingetragen. Die zu schreibenden Werte wurden von der Timing-Routine berechnet und in den Operanden der LDA-Befehle bei "HORIZO" und "VERTIC" eingetragen:

```
horizo  LDA #$00        ;Versch. vom linken
        STA $D016       ;Bildrand vertic
        LDA #$00        ;Versch. vom oberen
        STA $D011       ;Bildrand
        LDX #$57        ;Verzögerungsschleife
w1      DEX             ;bis Bildanfang
        BNE w1
        LDY #$17        ;Video-RAM bei $0400
        STY $D018       ;einschalten
        LDA $D011       ;$D011 laden
        AND #$1F        ;relev. Bits ausmaskieren
        STA $D011       ;und schreiben
        LDA #$0E        ;Bildschirmfarben
        STA $D020       ;zurücksetzen
        LDA  #$06
        STA $D021
        PLA             ;Prozessorregister vom
        TAY             ;Stapel holen und IRQ
        PLA             ;beenden.
        TAX
        PLA
        RTI
```

Soviel nun also zu unserer IRQ-Routine.
Sie erledigt für uns die entsprechenden Aufgaben zur Bildverschiebung. Allerdings muß sie auch irgendwie mit Basiswerten gefüttert werden, um das Timing für alle Fälle genau aufeinander anzustimmen. Dies übernehmen weitere Steuerroutinen, die wir im nächsten, und dann auch letzten Teil dieses Kurses besprechen werden.

(ub)

Teil 17 – Magic Disk 03/95

Herzlich Willkommen zum 17. und letzten Teil unseres IRQ-Kurses. In dieser Ausgabe möchten wir Ihre Ausbildung abschließen und Ihnen die letzten Geheimnisse der AGSP-Routine erläutern, mit deren Besprechung wir ja schon im letzten Monat begannen.
Dort hatten wir zunächst die IRQ-Routine besprochen, die es uns ermöglicht, den AGSP-Effekt durchzuführen. Sie bestand aus einer geschickten Kombination der Routinen FLD, VSP und HSP, sowie einigen Befehlen zum Softscrollen des Bildschirms. Zudem hatten wir noch einen einfachen Sprite-Multiplexer mit integriert. Wie Sie also sehen, ist die AGSP-Routine auch ein gutes Beispiel dafür, wie die Raster-IRQ- Effekte in Kombination miteinander ein perfektes Zusammenspiel ergeben können.
Prinzipiell ist die im letzten Monat besprochene AGSP-IRQ- Routine also in der Lage, uns den Bildschirm um jeden beliebigen Offset in Xund Y-Richtung zu verschieben (in X-Richtung um 0-319, in Y-Richtung um 0-255 Pixel) . Damit wir jedoch eine SPEZIELLE Verschiebung umsetzen können, muß die AGSP-Routine natürlich auch mit speziellen Parametern gefüttert werden, die von einer anderen Routine in der Hauptschleife des Prozessors vorberechnet werden müssen. Diese Routine existiert natürlich auch in unseren Programmbeispielen "AGSP1" und "AGSP2", sie heißt "Controll" und soll das

Thema dieses Kursteils sein.

1. Die Parameter AGSP-Routine

Wollen wir zunächst einmal klären, welche Parameter unsere AGSP-Routine benötigt. Da es eine IRQ-Routine ist, können wir selbige natürlich nicht so einfach übergeben. Sie müssen teilweise in Zeropageadressen oder sogar direkt im Code der IRQ-Routine eingetragen werden. Insgesamt gibt es 5 Stellen, die wir ändern müssen, damit die AGSP-Routine auch die Parameter bekommt, die sie benötigt, um eine ganz bestimmte Verschiebung durchzuführen. Die Verschiebungen in Xund Y-Richtungen werden wir im folgenden mit XPos und YPos bezeichnen. Unsere Beispielprogramme verwalten auch gleichnamige Variablen in der Zeropage. Sie werden von der Joystickabfrage automatisch auf die zwei benötigten Werte gesetzt, um die die AGSP-Routine den Bildschirm in beide Richtungen verschieben soll. Da dies in jedem Frame erneut geschieht wird so der Eindruck einer flüssigen Verschiebung erzielt. Da die X-Verschiebung Werte größer als 256 aufnehmen muß, benötigen wir für den Wert XPos also zwei Speicherstellen, die von unserem Beispielprogramm in den Zeropageadressen $02/$03 in Lo/ Hi-Byte-Darstellung verwaltet werden. Für YPos genügt uns ein Byte, das in der Zeropageadresse $04 zu finden ist.
Wollen wir nun jedoch zunächst herausstellen, welche Parameter aus diesen beiden Werten berechnet werden müssen, damit die AGSP-Routine die korrekte Bildschirmverschiebung durchführt:

1a. FLDCNT für FLD und VSP

Wie Sie sich sicherlich noch erinnern, so benutzte unsere AGSP-Routine eine FLD-Routine, um das Timing zur VSP-Routine auszugleichen, für den Fall, daß letztere weniger als 25 Charakterzeilen vertikale Bildschirmverschiebung durchführen sollte. Um nun die korrekte Anzahl Zeilen zu verschieben, hatte sich die FLD-Routine eines Zählers bedient, der ihr angab, wie viel Mal sie durchzulaufen hatte. Zur besseren Erläuterung des Zählers hier noch einmal der Kern der FLD-Routine:

```
fld     LDX #$27        ;Max. Anz. FLD-Durchl.
        LDX #$01        ;Index $D011/$D018-Tabelle
fldlp   JSR cycles      ;12 Zyklen verzögern
        LDA field1,Y    ;Wert für $D011 holen
        STA $D011       ;und setzen
        LDA field2,Y    ;Wert für $D018 holen
        STA $D018       ;und setzen
        NOP             ;6 Zyklen verzögern
        NOP
        NOP
        INY             ;Tab-Index+1
        DEX             ;FLD-Durchläufe-1
        CPX <fldcnt     ;=erforderliche Anzahl
        BNE fldlp       ; Durchl.?Nein→Weiter
```

Der besagte FLD-Zähler befindet sich nun im X-Register, das mit dem Wert $27 vorinitialisiert wird. Dieser Wert ergibt sich aus der Anzahl Rasterzeilen, die für das VSP-Timing benötigt werden. Normalerweise sollten dies 25 Rasterzeilen sein. Da wir jedoch ein Spritescoreboard mit einer Gesamthöhe von 42(=$2A) Rasterzeilen darstellen, muß auch diese Anzahl Zeilen übersprungen werden. Der FLD-Zähler enthält diesen Wert minus 3, da ja auch schon durch die IRQ-Glättung 2 Rasterzeilen verbraucht wurden und nach der FLD-Routine der Zähler wieder für die VSP-Routine um 1 erhöht wird.
Dieser Zähler im X-Register wird nun pro Rasterzeile einmal herabgezählt, solange, bis er dem Wert in FLDCNT entspricht, womit die FLD-Schleife verlassen wird. Aus dem

verbleibenden Wert im X-Register, der dann genau dem FLDCNT-Wert entspricht, ergibt sich der VSPCNT, der angibt, wie oft die VSP-Routine durchlaufen werden muß. Dieser Zähler muß nicht eigens berechnet werden. Somit ist VSPCNT also einer der 5 Parameter, die die AGSP-Routine benötigt. Er berechnet sich aus YPos dividiert durch 8.

1b. REDU1 und REDU2 für HSP

Nachdem also der Parameter für die VSP- Routine und somit der Charakterzeilen-Verschiebung geklärt ist, wollen wir zu den Parametern für die HSP-Routine kommen. Dies sind keine Parameter im eigentlichen Sinne, sondern Änderungen im Code der HSP-Routine, die eine Verzögerung der gewünschten Dauer durchführen.
Wie Sie sich vielleicht erinnern, so wird mit jedem Taktzyklus, den die HSP-Routine das Zurücksetzen des Bildschirms auf "Charakterzeile zeichnen" verzögert, die Darstellung desselben um ein Zeichen (also 8 Pixel) nach rechts verschoben.
Das heißt also, daß wir XPOS/8 Taktzyklen verzögern müssen, um die gewünschte Verschiebung zu erzielen. Auch hier zeige ich Ihnen wieder einen Auszug aus der HSP-Routine, der der besseren Erläuterung dienen soll:

```
HSP     LDX field1+3,Y    ;nächsten $D011-Wert
        STX $D011         ;schreiben
        JSR cycles        ;Anfang sichtbaren Bildschirm abwarten
        DEX               ;$D011-Wert-1
redu1   BEQ redu2         ;Ausgleich für unge-
redu2   BNE tt            ;rade Zyklen
        NOP               ;Insgesamt 20
        …                 ;NOPs für das
        NOP               ;HSP-Timing
tt      STX $D011         ;akt.Z. einschalt.
```

Die HSP-Routine setzt die Y-Verschiebung des Bildschirms nun zunächst hinter den Rasterstrahl, um so dem VIC vorzugaukeln, er befände sich noch nicht in einer Rasterzeile. Anschließend wird mit dem JSR-, dem DEX-, und den BEQ-/BNE-Befehlen auf den Anfang des linken Bildschirmrandes gewartet, wobei die letzten beiden Befehle ausschließlich für das taktgenaue HSP-Timing herangezogen werden. An den Labels "REDU1" und "REDU2", wird unsere Parameterberechnungsroutine später den Code modifizieren, so daß das Timing exakt dem gewünschten XPos/8- Offset entspricht. Hierzu dienen auch die 20 NOP-Befehle, die genau 40 Zyklen, bzw. Zeichen verzögern, wodurch im Extremfall alle Zeichen übersprungen werden können. Der anschließende STX-Befehl setzt dann die Darstellung wieder auf Charakterzeilenbeginn, womit der VIC dazu gezwungen wird, sofort eine Charakterzeile zu lesen und anzuzeigen. Also erst hier wird der eigentliche Effekt ausgeführt. Um nun das Einsetzen der Charakterzeile genau abzutimmen, müssen wir also um XPOS/8 Taktzyklen verzögern.
Diese Verzögerung kann recht haarig umzusetzen sein, wenn es sich dabei um eine ungerade Anzahl Zyklen handelt.
Diese Aufgabe soll der Branch-Befehl, beim Label REDU1 lösen. Zunächst einmal sollte erwähnt werden, daß das X-Register nach dem DEX-Befehl IMMER einen Wert ungleich null enthält, da dort ja der benötigte Wert für $D011 steht, der immer größer als 1 ist, womit durch den DEX-Befehl nie auf 0 herabgezählt werden kann. Das Zeroflag ist beim Erreichen des Labels REDU1 also immer gelöscht.
Wie auch schon beim Glätten des IRQs benutzen wir nun also den Trick mit den Branch-Befehlen, um ggf. eine ungerade Verzögerung zu erzielen. Wie Sie von dort vielleicht noch wissen, benötigt ein Branch-Befehl immer mindestens 2 Taktzyklen. Trifft die abgefragte Bedingung nun zu, so dauert die Abarbeitung des Branchbefehls immer einen Zyklus mehr, also 3 Zyklen. Soll der Bildschirm nun um eine gerade Anzahl Zeichen nach rechts

verschoben werden, so trägt die Parameterberechnungsroutine bei REDU1 den Assembler-Opcode für einen BEQ-Befehl ein. Da diese Bedingung nie erfüllt ist, werden nur 2 Zyklen verbraucht und direkt mit dem folgenden Befehl bei REDU2 fortgefahren. Muß eine ungerade Anzahl Taktzyklen verzögert werden, so setzt die Parameterberechnung bei REDU1 den Opcode für einen BNE-Befehl ein. Da diesmal die Bedingung zutrifft, dauert die Abarbeitung des Befehls 3 Taktzyklen, wobei jedoch ebenfalls mit REDU2 fortgefahren wird. Dadurch dauert die Routine nun einen Taktzyklus mehr, womit wir eine ungerade Verzögerung erzielt haben.
Der Branch-Befehl bei REDU2 muß nun ebenfalls modifiziert werden. Hierbei bleibt der Opcode jedoch immer derselbe, also ein BNE-Befehl, der immer wahr ist.
Es wird lediglich der Operand dieses Befehls geändert, so daß nicht mehr auf das Label "TT" gesprungen wird, sondern auf einen der zuvor stehenden NOP-Befehle, wodurch die entsprechende HSP-Verzögerung sichergestellt wird. Der Operand eines Branch-Befehles kann nun ein Bytewert zwischen -127 und +128 sein, der den Offset angibt, um den der Branch-Befehl den Programmzähler des Prozessors erhöhen, bzw. erniedrigen soll. Steht hier also der Wert $05, so werden die nächsten 5 Bytes im Code übersprungen. Da ein NOP-Befehl immer ein Byte lang ist, werden also in unserem Fall exakt 5 NOPs übersprungen, womit noch 15 NOPs ausgeführt werden, die 15*2=30 Taktzyklen verzögern, und somit den Bildschirm in der Horizontalen erst ab dem dreißigsten Charakter beginnen lassen. Der Sprungoffset für den BNE-Befehl berechnet sich danach also aus der Formel: 'Anzahl NOPs'-XPOS/8/2=20-XPOS/16.
Mit Hilfe dieser beiden Branch-Befehle hätten wir nun also das entsprechende Timing für die HSP-Routine korrekt umgesetzt. Beachten Sie nur bitte noch folgenden Umstand: Dadurch, daß der Branch-Befehl bei REDU2 immer ausgeführt werden muß, also daß seine Bedingung immer wahr sein muß, müssen wir hier einen BNE-Befehl verwenden. Gerade aber WEIL dessen Bedingung immer wahr ist, benötigt er auch immer 3 Taktzyklen, also eine ungerade Anzahl, womit er den Ausgleich des Branch-Befehls bei REDU1 wieder zunichte machen würde. Damit dies nicht geschieht, muß die Parameterumrechnungsroutine die Opcodes genau umgekehrt einsetzen, wie oben beschrieben:
also einen BNE-Befehl, wenn eine gerade Anzahl Zyklen verzögert werden soll, und einen BEQ-Befehl, wenn die Anzahl der zu verzögernden Zyklen ungerade ist! Ich erläuterte dies im obigen Fall zunächst anders, um Sie nicht noch zusätzlich zu verwirren!

1c. Parameter für das Softscrolling

Damit wären also alle Parameterberechnungen, die Raster-IRQ-Effekte betreffend, abgehandelt. Da HSP und VSP-Routine den Bildschirm jedoch immer nur in 8-Pixel-Schritten verschieben, müssen wir diesen nun noch mit den ganz normalen Softscroll-Registern um die fehlende Anzahl Einzelpixel verschieben. Auch dies wird noch innerhalb des AGSP-Interrupts durchgeführt, nämlich genau am Ende desselben. Hier hatten wir zwei Labels mit den Namen "HORIZO" und "VERTIC" untergebracht, die jeweils auf eine LDA-STA- Befehlsfolge zeigten, und den entsprechenden Verschiebeoffset in die VIC-Register $D011 und $D016 eintrugen.
Hier nochmal ein Codeauszug aus dem AGSP-IRQ:

```
horizo      LDA #$00        ;Versch. vom linken
            STA $D016       ;Bildrand
vertic      LDA #$00        ;Versch. vom oberen
            STA $D011       ;Bildrand
```

Um nun die Werte dieser Verschiebungen zu ermitteln, müssen wir lediglich jeweils die untersten 3 Bits aus XPos und YPos ausmaskieren und in die Oparanden-Bytes der LDA-

Opcodes eintragen. Da die Register $D011 und $D016 jedoch auch noch andere Aufgaben erfüllen, als lediglich das Softscrolling des Bildschirms zu setzen, müssen auch zur korrekten Bildschirmdarstellung notwendige Bits in diesen Registern mitgesetzt werden. Auch dies wird unsere Parameterberechnungsroutine übernehmen.

2. Die AGSP-Parameter-Routine "CONTROLL"

Kommen wir nun endlich zu der Unterroutine, die die benötigten Parameter in umgerechneter Form in die AGSP-Routine einbettet, und letztere zu einer korrekten Anzeige des Bildschirms bewegt.
Nachdem wir die Funktionprinzipien der Parameterübergabe nun ausführlich besprochen haben, besteht die Controll-Routine nur noch aus ein paar " Rechenaufgaben". Sie wird von der Border-IRQ- Routine aufgerufen und wertet die Einträge in XPos sowie YPos aus. Diese Werte werden, wie schon erwähnt, von der Joystickabfrage in der Hauptschleife unseres Programmbeispiels entsprechend gesetzt und verwaltet.
Dadurch, daß Controll während des Border- IRQs aufgerufen wird, stellen wir gleichzeitig auch sicher, daß die AGSP-Routine zu einem Zeitpunkt modifiziert wird, zu dem sie nicht ausgeführt wird, und vermeiden so ihr Fehlverhalten.
Hier nun also die Controll-Routine, die als erstes die erforderlichen Werte für die Y-Verschiebung berechnet. Dies ist zunächst der Wert für FLDCNT, gefolgt von dem entsprechenden Eintrag für die vertikale Softscrolling-Verschiebung, die in VERTIC+1 eingetragen werden muß.
FLDCNT berechnet sich einfach aus YPos/8. Der Softscrollwert entspricht den untersten 3 Bits von YPos, also (YPos AND $07). Hierbei muß jedoch durch das HSP-Timing der benötigte Wert minus 1 ins Softscrollregister eingetragen werden. Das hängt damit zusammen, daß die Softscrollveränderung ja ebenfalls in Register $D011 eingetragen werden muß, was ja zugleich Dreh- und Angelpunkt aller anderen Raster-IRQ- Routinen ist. Die gewünschte Änderung führen wir mit Hilfe einer speziellen Befehlsfolge durch, um uns Überlaufvergleiche zu sparen. Zum Schluß steht in jedem Fall der richtige Wert im Akku, in den wir dann mittels ORA-Befehl noch die Bits 3 und 4 setzen, die den 25 Zeilen-Schirm, sowie den Bildschirm selbst einschalten, was bei $D011 ja ebenfalls noch berücksichtigt werden muß:

```
Controll    LDA <ypos       ;YPOS holen
            LSR             ;durch 8
            LSR             ;dividieren
            LSR
            STA <fldcnt     ;und in FLDCNT ablegen
            CLC             ;C-Bit für Addition löschen
            LDA <ypos       ;YPos holen
            AND #$07        ;Unterste Bits ausmaskieren
            EOR #$07        ;umkehren
            ADC #$1A        ;Ausgleichswert addieren
            AND #$07        ;wieder maskieren
            ORA #$18        ;Bilschirmbits setzen
            STA vertic+1    ;und in AGSP ablegen
```

Der ORA-Befehl am Ende ist übrigens von wichtiger Bedeutung. Wie Sie ja wissen, unterscheiden sich die beiden Beispiel "AGSP1" und "AGSP2" nur darin, daß die erste Version einen Textbildschirm, die zweite Version einen Multicolor-Grafik-Schirm scrollt. Der programmtechnische Unterschied zwischen diesen beiden Routinen besteht nun lediglich in der Änderung des oben aufgeführten ORA-Befehls. In AGSP2 wird hier mit dem Wert $78 verknüpft, womit zusätzlich auch noch Hires- und Extended-Background-Color-Mode eingeschaltet werden. Dies ist der EINZIGE Unterschied zwischen den beiden Beispielen, der eine große Auswirkung auf das Erscheinen hat! Die AGSP-Routine selbst ändert sich

durch die Grafikdarstellung nicht!

Als nächstes behandelt die Routine den XPos-Wert. Hierzu legt sie sich zunächst eine Kopie von Low- und Highbyte in den Labels AFA und AFB an, die den Zeropageadressen $07 und $08 zugewiesen sind.
Anhand des 3. Bits aus dem Low-Byte von XPos kann ermittelt werden, ob eine gerade (Bit gelöscht) oder ungerade (Bit gesetzt) Anzahl Taktzyklen bei HSP verzögert werden muß. In letzterem Fall muß in REDU1 der Opcode für einen BEQ-Befehl eingesetzt werden. Im anderen Fall steht dort ein BNE-Opcode, der ganz am Anfang des folgenden Sourcecode-Segmentes dort eingetragen wird:

```
                LDX #$D0        ;Opcode für "BNE" in
                STX redu1       ;REDU1 eintragen
                LDA <xpos+1     ;High-Byte XPos holen
                STA <afb        ;und nach AFB kopieren
                LDA <xpos       ;Low-Byte XPos holen
                STA <afa        ;nach AFA kopieren
                AND #$08        ;3. Bit ausmaskieren
                BNE co1         ;<>0-> alles ok, weiter
                LDX #$F0        ;Sonst Opcode für "BEQ"
                STX redu1       ;in REDU1 eintr.
```

Nachdem der einzelne Taktzyklus korrigiert wurde, müssen wir nun noch den Sprungoffset für den BNE-Befehl bei REDU1 ermitteln. Hierzu wird die Kopie des 16- Bit-Wertes von XPos zunächst durch 16 dividiert, das daraus resultierende 8- Bit Ergebnis muß nun noch von dem Wert 20 (der Anzahl der NOPs in der HSP-Routine) subtrahiert werden, bevor es in REDU2+1 abgelegt werden kann:

```
co1             LSR <afb        ;Wert in AFA/AFB 4 Mal
                ROR <afa        ;nach rechts rotieren
                LSR <afb        ;(=Division durch 16)
                ROR <afa
                LSR <afb
                ROR <afa
                LSR <afb
                ROR <afa
                SEC             ;C-Bit für Subtraktion setzen
                LDA #$14        ;Akku=20 NOPS
                SBC <afa        ;Ergebnis subtrahieren
                STA redu2+1     ;und ablegen
```

Folgt nun nur noch die Berechnung des horizontalen Softscrollwertes. Dieser entspricht den untersten 3 Bits von XPos. Da wir auch hier ein Register ($D016) beschreiben, das auch noch andere Aufgaben erfüllt, müssen wir in Ihm ebenfalls noch ein Bit seten, nämlich das dritte, daß die 40 Zeichen/ Zeile-Darstellung einschaltet. In AGSP2 setzen wir hier zusätzlich auch noch das vierte Bit, daß den Multicolor-Modus einschaltet:

```
                LDA <xpos       ;XPos holen
                AND #$07        ;unterste 3Bits ausmaskieren
                ORA #$08        ;40 Zeichen/Zeile
                STA horizo+1    ;und ablegen
                RTS             ;ENDE
```

Das ist nun alles wichtige gewesen, was noch zu AGSP zu sagen war. Wir sind damit auch wirklich beim allerletzten Teil unseres Kurses über die Raster-IRQ-Programmierung

angelangt. Ich hoffe es hat Ihnen die letzten 17 Monate viel Spaß bereitet, in die tiefsten Tiefen unseres "Brotkasten" vorzudringen,"where (almost) no man has gone before"! Einen besonderen Dank möchte ich auch meinem Kollegen Ivo Herzeg (ih) aussprechen, ohne dessen unermüdlichen Bemühungen um exaktes Rastertiming dieser Kurs bestimmt nicht zustande gekommen wäre.

(ih/ub