Assembler ist keine Alchimie (Teil 7)

In diesem Teil werden wir drei Themen besprechen: den Stapel und die indirekte Adressierung. Ferner zeigen wir an einem ausführlichen Beispiel, wie Text auf dem Bildschirm und auf dem Drucker ausgegeben werden kann.

Die Assembler-Folge in der letzten Ausgabe hat einige Fragen offengelassen, die dieses Mal beantwortet werden sollen: Die Sache mit dem Stapel wird geklärt, und Sie werden Befehle zu gebrauchen lernen, die uns Stapeloperationen ermöglichen. In unserem ersten Programmprojekt-Teil gab es für Sie unverständliche Sequenzen, die mit der Speicherstelle 1 zu tun hatten. Auch das werden Sie diesmal verstehen.

Sie beherrschen inzwischen fast alle Arten der Adressierung: Nach dieser Folge fehlt keine mehr. Ich habe Ihnen hoffentlich das Wasser im Munde zusammenlaufen lassen mit der Liste aller Kernal-Routinen: Nun sollen uns die ersten davon munden.

Wir stapeln

In der vorangegangenen Ausgabe haben wir beim JSR-Befehl schon den Stapel etwas kennengelernt. Aber so ganz genau wissen wir's ja noch nicht, was das ist. Deswegen jetzt mal im Detail: Der Stapel, auch Prozessorstack genannt, ist der Speicherbereich von dezimal 256 ($100) bis dezimal 511 ($1FF), der direkt von unserer CPU verwaltet wird. Das ist also die gesamte Page 1. Ähnlich wie bei der String-Verwaltung geschieht auch hier das Füllen von oben nach unten. Das erste Byte, welches in den Stack geschoben wird, kommt also nach $1FF, das nächste nach $1FE und so weiter. Voll ist der Stapel, wenn auch $100 besetzt wurde (siehe Bild 1).

Warum heißt das Ding nun eigentlich Stapel? Das erklärt sich aus dem Zugriffs-Prinzip. Man spricht von einer LIFO-Struktur, von »**L**ast **I**n — **F**irst **O**ut«, zu deutsch »zuletzt hinein — zuerst heraus«. Das zuerst hineingebrachte Byte befindet sich am Speicherboden ($1FF), das zuletzt eingebrachte an der Speicherspitze. Stellen Sie sich einen Stapel Akten vor (Bild 2).

Offensichtlich wurde der 4. Aktenordner zuletzt auf den Stapel gesteckt. Er kann zuerst heruntergeholt werden. An die Akte 1 kommen wir erst heran, wenn alle anderen heruntergenommen worden sind. Genauso verhält es sich mit dem Prozessorstack: Um an das unterste Byte des Stapels heranzukommen, müssen erst Byte für Byte die darüberliegenden (nach Bild 1 eigentlich die darunterliegenden) weggeschafft werden.

Mit dem Prinzip des Stapelspeichers werden Sie sich auskennen, wenn Sie schon mal andere Programmiersprachen als Basic ausprobiert haben: In Forth beispielweise operieren Sie ständig mit Stapeln.

Der Stapel: Das Gedächtnis des Prozessors

Damit wir — und der Prozessor — den Überblick über den Stack behalten, gibt es dankenswerterweise noch einen Stapelzeiger (stackpointer), der jeweils auf den nächsten freien Platz des Stapels weist. Da gibt's nun aber ein kleines Problem: Der Stapel belegt die komplette Seite 1.

Ein Stapelzeiger, der auf zum Beispiel $01FE zeigen soll, müßte das MSB (also 01) und das LSB (also FE) in zwei Bytes lagern. Der Stapelzeiger ist aber nur 8 Bit groß ... Freundlicherweise sorgt unser Mikroprozessor automatisch für das neunte Bit. Der Zeiger zählt also immer von $FF an rückwärts bis $00 und weist dabei von $1FF bis $100.

Der Stack hat in unserem Computer drei Aufgaben zu erfüllen:

1) Organisation von Unterprogramm-Adressen
2) Zwischenspeicherung bei Unterbrechungen (Interrupts)
3) vorübergehende Datenspeicherung

Die Rolle des Stapels bei Unterprogramm-Aufrufen haben wir in der letzten Folge schon ausgiebig behandelt. Die sogenannten Interrupts heben wir uns noch für später auf — dazu fehlen uns noch ein paar Kenntnisse. Mit der vorübergehenden Speicherung von Daten befassen wir uns gleich, wenn wir an die Befehle zur Stackbehandlung herangehen.

Zuvor — weil das hier gerade ganz gut paßt — noch ein paar Gedanken zur rekursiven Programmierung. Gemeint ist damit eine Programmstruktur, in der sich ein Unterprogramm selbst aufruft. Auch GOSUB-Befehle in Basic bewirken Einträge der Rücksprungadressen im Stapel. Auf diese Weise ergibt sich für unseren Computer eine begrenzte Verschachtelungstiefe bei Unterprogrammaufrufen. Diese wird bei Rekursion besonders schnell erreicht, und das bewirkt die Ausgabe einer OUT OF MEMORY-Fehlermeldung.

Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS

Mit dem Stapel haben wir 256 Speicherplätze für eine schnelle Zwischenspeicherung aller möglichen Daten zur Verfügung. Weil der 6510 (und natürlich auch der 6502) diesen Speicherbereich wie die Zeropage behandelt, geht das Speichern sehr schnell. Man muß nur immer die spezielle LIFO-Struktur berücksichtigen.

Im Grunde braucht man eigentlich nur zwei Befehle: Etwas auf den Stapel schieben (in der Literatur oft als Push-Befehl bezeichnet) und etwas herunterziehen, das nennt man dann Pull- oder auch Pop-Befehl.

Unser Prozessor kennt insgesamt sechs auf den Stapel wirkende Anweisungen:

PHA Damit schreibt man den Akku-Inhalt in den Stapel (»push-accumulator«). Der Stapelzeiger wird automatisch eine Position heruntergezählt (er rechnet ja von $FF an abwärts!). Der Inhalt des Akku wird dabei nicht verändert. Deswegen bleibt auch das Status-Register (also die ganzen Flaggen: N V B D I Z C) unbeeinflußt.

PLA »Pull accumulator«. Das ist der umgekehrte Weg: Das, was zuoberst auf dem Stapel liegt, wird in den Akku geschrieben. Dadurch wird ein Stapelplatz frei, was den Stapelzeiger veranlaßt, um 1 zu wachsen. Weil das, was da in den Akku geladen wird, 0 sein kann oder auch negativ (also mit gesetztem Bit 7), wird unter Umständen auch die N- oder die Z-Flagge verändert.

Weniger mit Datenzwischenspeicherung haben die anderen Befehle zur Stapel-Manipulation zu tun:

PHP Das steht für »push processor status«, also »schiebe das Prozessor-Status-Register auf den Stapel«. Der aktuelle Flaggenstand kann damit aufbewahrt werden. Das Status-Byte ändert seinen Inhalt dabei ebensowenig wie der Akku bei PHA. Auch hier wird der Stapelzeiger freundlicherweise um 1 herabgezählt.

PLP »Pull processor status«, »hole den Prozessor-Status vom Stapel«, ist der umgekehrte Befehl, der (wie bei PLA in den Akku) den Wert, der zuoberst im Stapel liegt, in das Flaggen-Register schreibt. Da sollte man höllisch aufpassen, was man damit einlädt: Das ist eine feine Gelegenheit für den Computer abzustürzen. Der Stapelzeiger wird — wie gehabt — um 1 erhöht.

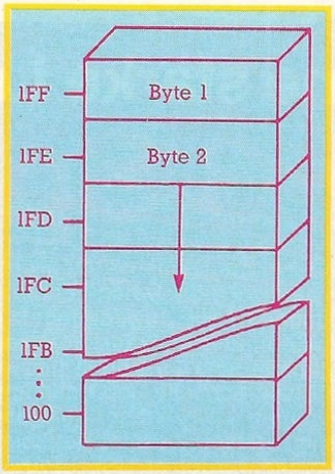

Bild 1. So wird der Stapel gefüllt

Nicht direkt mit dem Stapel, sondern mit dem Stapelzeiger befassen sich die beiden folgenden Befehle:

TSX »Transfer stack-pointer into X«, zu deutsch, »schiebe den Stapelzeiger ins X-Register« eröffnet die Möglichkeit, den Stapelzeiger zu lesen. Dabei bleibt er selbst unverändert erhalten. Weil nun im X-Register alle Werte zwischen $FF und 0 auftreten können, werden auch die Flaggen beeinflußt (N- und Z-Flagge).

TXS Den umgekehrten Weg geht »transfer X into stackpointer« = »übertrage X-Register-Inhalt in den Stapelzeiger«. Das ist der einzige Befehl, der es erlaubt, den Stapelzeiger mit einem von uns kontrollierten Wert zu laden. Der Inhalt des X-Registers bleibt dabei unverändert, demzufolge interessieren sich auch die Flaggen nicht dafür.

Alle sechs Anweisungen bestehen nur aus einem Byte und sind implizit adressiert. Die Stapelzeiger-Befehle TXS und TSX benötigen zwei Taktzyklen, die Push-Befehle je drei und die Pull-Befehle vier Taktzyklen zur Bearbeitung.

Es ist etwas schwierig, Stapel-Operationen direkt zu verfolgen. Die meisten Assembler — so anscheinend auch der SMON — gebrauchen ebenfalls diesen Speicherbereich. Verlangt man beispielsweise mit dem SMON-Kommando M 0100 01FF eine Darstellung des Stapelinhaltes, dann findet man eine ganze Menge Spuren der Arbeit des Assemblers. Versucht man die zu löschen oder zu überschreiben, zum Beispiel mit dem nachfolgenden kleinen Programm, dann hat der Assembler die Mühe schon wieder zunichte gemacht, wie man durch erneutes M 0100 01FF schnell sehen kann. Dieses kleine Programm soll unterhalb des durch den Stapelzeiger bezeichneten Bereichs 32 Nullen in den Stapel schreiben:

```
8000    LDA #00
8002    TSX
```

Der Stapelzeiger wird ins X-Register gerettet.

```
8003    LDY #20
8005    PHA
```

Wir schieben eine Null auf den Stapel.

```
8006    DEY
8007    BNE 8005
8009    TXS
```

Nach 32 Eintragungen von Nullen stellen wir den alten Stapelzeiger wieder her.

```
800A    BRK
```

Erneutes Kommando M 0100 01FF zeigt keine Nullen. Erst wenn wir anstelle des TXS in Zeile 8009 ein BRK schreiben, den Stapelzeiger also nicht zurückschreiben, erscheinen unsere Nullen. Sieht man genau hin, dann stellt man fest, daß unterhalb des durch den Stapelzeiger bezeichneten Bereichs genau der gleiche Inhalt zu finden ist wie vorher, nur eben mit dem Stapelzeiger verschoben.

Ganz konnte ich dies Rätsel noch nicht lösen, muß ich gestehen, aber für den Gebrauch des Stapels ändert sich dadurch für uns nichts. Worauf muß man achten bei Stapeloperationen? Ganz einfach: Zwischen dem Ablagern eines Wertes auf dem Stapel und dem Zurückholen muß für jeden Push-Befehl ein Pull-Befehl vorhanden sein, für jedes weitere PHA ein PLA, für jedes JSR ein RTS. Nur wenn wir auf diese Symmetrie der Push- und der Pull-Befehle achten (und wie Sie noch aus der vorhergegangenen Ausgabe wissen, sind ja JSR und RTS ebenfalls dazuzurechnen), können wir sicher sein, daß der Stapelzeiger zum Zeitpunkt des Rückholens eines Wertes vom Stapel auch wirklich darauf deutet. Wenn man also nicht ganz genau weiß, wie der verwendete Assembler den Stapel nutzt, sollte man auf Operationen mit den Befehlen TSX und TXS verzichten.

Nun können Sie schon einen Teil der bislang unbekannten Programmsequenz aus der letzten Folge verstehen. Im zweiten Programmteil hatten wir mit

```
02CE    LDA 01
02D0    PHA
```

den Inhalt der Speicherstelle 01 in den Akku geladen und auf den Stapel geschoben. Später — nach einigen weiteren Operationen — wurde dann dieser Speicherinhalt wiederhergestellt durch

```
02E7    PLA
02E8    STA 01
```

Was aber hat es mit dieser Speicherstelle 01 auf sich? Das soll nun als nächstes erklärt werden.

Sein oder Nichtsein: Das Rätsel des Prozessorports

Der Commodore 64 hat 64 KByte an RAM zu bieten. Außerdem aber verfügen wir beim normalen Programmieren über weitere 24 KByte, in denen das Betriebssystem, der Basic-Interpreter, Ein- und Ausgabebausteine und der Zeichenspeicher stecken. Wie Sie aus der ersten Assemblerfolge wissen, umfaßt der Adreßbus aber nur 16 Bits, was uns lediglich 65536 Speicherzellen, also 64 KByte adressieren läßt. Des Rätsels Lösung liegt darin, daß einige Adressenbereiche mehrfach belegt sind. Man kann das vergleichen mit dem Trick des Kastens mit dem doppelten Boden. Welcher Kasteninhalt gerade dem Prozessorzugriff offensteht, wird durch den Prozessorport, das sind die Speicherstellen 00 und 01, gesteuert.

Dr. Helmuth Hauck hat in seiner Serie »Memory Map mit Wandervorschlägen« (64'er, Ausgabe 11 (1984), Seite 135 ff.) die genaue Funktion jedes Bits dieser beiden Speicherstellen erklärt. Wer noch mehr wissen möchte — auch über die Wirkungsweise der beiden Leitungen »Game« und »Exrom« — sollte das nachlesen im »Commodore 64 Programmers Reference Guide« ab Seite 260. Für uns als angehende Assembler-Alchimisten ist die Speicherstelle 1 aber so wichtig, daß wir ganz kurz hier nochmal darauf eingehen.

Die Speichersteuerfunktionen haben die Bits 0 bis 2 der Speicherstelle 1. Je nach Belegung dieser Bits gestaltet sich die 64-KByte-Landschaft unseres Computers wie in Tabelle 1 gezeigt.

Was können wir als Maschinen-Programmierer mit dieser Kenntnis anfangen? Theoretisch stehen uns für unsere Programme damit 64 KByte offen. Praktisch werden wir nur in den seltensten Fällen auf die Ein- und Ausgabe-Bausteine verzichten können. Lassen wir ein reines Maschinenprogramm laufen, ohne jeglichen Rückgriff auf Interpreter oder Betriebssystem, dann haben wir immerhin noch zirka 60 KByte zur freien Verfügung. Benutzen wir Routinen aus diesen beiden ROM-Bausteinen, dann müssen wir sie allerdings — zumindest für den Zeitpunkt des Routineaufrufs — wieder einschalten. Wenn wir — was wohl meistens der Fall sein wird — Kombinationen von Basic- und Assemblersprache verwenden, können wir den gesamten Basic-Speicher bis $A000 frei halten, können auch den bei allen Beispielprogrammen so beliebten Bereich $C000 bis $D000 leer lassen und packen unsere Routinen weitgehend unter die ROMs, die dann jeweils beim Aufruf abgeschaltet werden. So haben wir eine Menge zusätzlichen Speicherplatz ergattert.

Nun können wir auch den letzten Rest des bislang unklaren Programms aus der letzten Folge verstehen. Nachdem wir den Inhalt der Speicherstelle 1 auf den Stapel gerettet haben (Zeilen $02CE und $02D0), schreiben wir $35 in den Prozessorport:

```
02D1    LDA #35
02D3    STA 01
```

$35 ist binär 0011 0101. Die Bits 0 bis 2, auf die es uns in diesem Zusammenhang ankommt, bewirken nun das Ausschalten des Interpreters und des Betriebssystems. Die Ein- und Ausgabe-Bausteine bleiben aktiv. Im weiteren Programmverlauf lesen wir die Speicherinhalte ab $E000, wobei wir nun den RAM-Inhalt erfassen. Das sollte vielleicht nochmal klargestellt werden: Jedes Hineinschreiben in die mehrfach belegten Speicherbereiche (dabei sind die Ein- und Ausgabe-Bausteine aber ausgenommen) wird automatisch in den RAM-Bereich umgelenkt. Das ist ja auch klar: In ein ROM kann eben nicht geschrieben werden. Deshalb braucht man dabei die ROMs

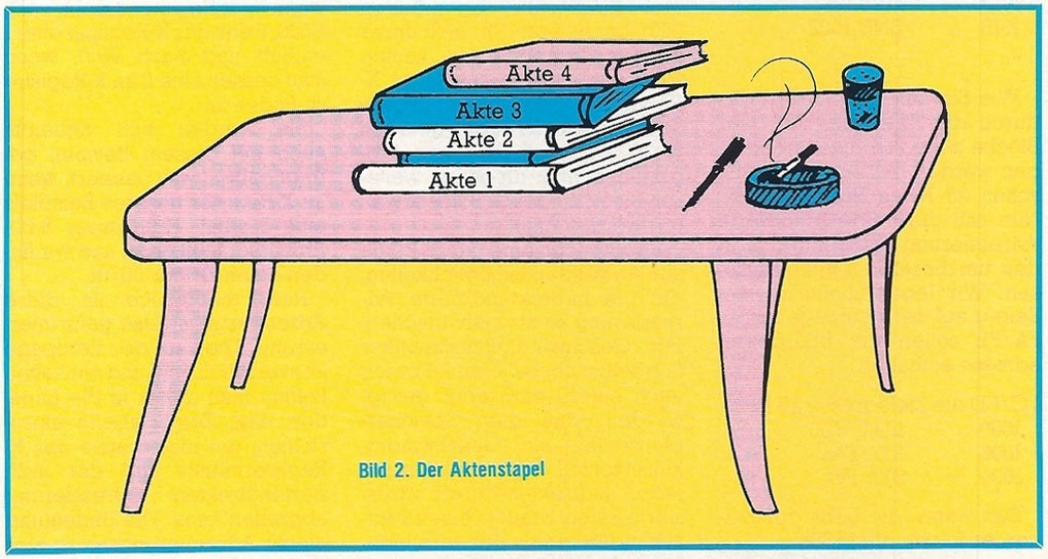

Bild 2. Der Aktenstapel

nicht auszuschalten. Jeder Lesevorgang greift aber auf die ROMs zu, weshalb man sie in unserem Fall ausschalten muß. Wie schon oben beim Stapel erklärt, schalten wir durch das Zurückholen des vorher dorthin geretteten alten Inhalts der Speicherstelle 1 in den Prozessorport wieder den Normalzustand ein.

Die wandernden toten Briefkästen der Assembler-Alchimisten

Wir werden nun die beiden letzten noch ausstehenden Arten der Adressierung kennenlernen. Beides sind sogenannte indirekte Adressierungsarten. Mit dem indirekten JMP-Befehl (zum Beispiel JMP(0300)) sind wir in der letzten Folge schon vertraut geworden. Wir hatten auch gelernt, daß es sich hierbei um einen absoluten Einzelgänger handelt, der nur für so einen Sprung erlaubt ist. Ebenso haben wir die indizierte Adressierung zu beherrschen gelernt: Das war die Sache mit den Indexregistern X oder Y. Eine Kombination aus beiden (also der indirekten und der indizierten) Adressierungsarten sind die indiziert-indirekte und die indirekt-indizierte Adressierung.

Die indirekt-indizierte Adressierung

Fangen wir mit der sehr häufig benutzten indirekt-indizierten Adressierung an: Man nennt sie auch »indirekt Y« oder »nachindizierte indirekte« Adressierung. Am besten sehen wir uns mal so einen Befehl an:

LDA (FA),Y

Die Klammer erinnert uns an den indirekten JMP-Befehl. Tatsächlich hat sie hier auch dieselbe Funktion: In FA und FB steht ein Zeiger auf eine Adresse. Nehmen wir mal an, die Belegung der Speicher wäre:

```
FA      01
FB      80
```

und im Y-Register stünde eine 5. Der Zeiger FA/FB weist also auf die Speicherstelle 8001. Da haben wir also wieder das Prinzip des toten Briefkastens. Der Computer guckt in den hohlen Baum FA/FB (LSB in FA, MSB in FB) und findet dort die Treffpunktadresse. Nun sind diese toten Briefkästen aber auch den gegnerischen Alchimisten-Agenten bekannt. Es kommt also noch ein Trick dazu: Zur dort aufgefundenen Adresse wird der Inhalt des Y-Registers addiert. In unserem Fall fanden wir also in FA/FB die Adresse 8001, im Y-Register steht eine 5, somit ist die endgültige Adresse 8001+5 = 8006. Unser Beispiel »LDA(FA),Y« bewirkt daher, daß in den Akku der Inhalt der Speicherstelle 8006 geladen wird. Nachindiziert nennen manche die Adressierung deswegen, weil zunächst dem Zeiger nachgegangen wird, der in unserem Beispiel auf 8001 weist, und erst danach durch Addition des Inhalts des Y-Registers die endgültige Speicherstelle (hier also 8006) berechnet wird.

Als Zeiger (also die Adresse in der Klammer) sind nur Zeropagespeicherstellen verwendbar, als Indexregister darf man hier nur das Y-Register gebrauchen. Von den bisher behandelten Befehlen können ADC, CMP, LDA, SBC und STA mit dieser Adressierungsart verwendet werden. Genaueres finden Sie wieder in der Tabelle mit der Befehlsübersicht (Tabelle 2).

Bevor wir uns dem anderen indirekten Adreß-Modus zuwenden, wollen wir uns überlegen, wozu man die indirekt-indizierte Adressierung verwendet. Wie Sie sich natürlich erinnern können, konnte man mit der normalen indizierten Adressierung, zum Beispiel mit

LDA 8000,Y

durch Variation des Indexregisters (hier das Y-Register) 256 Speicherstellen erfassen (Y von FF herunter bis 00). Will man mehr als diese 256 berücksichtigen, dann muß eine neue Basis (im Beispiel also anstelle der 8000) gewählt werden. Um das zu illustrieren, sehen wir uns mal den Anfang eines Programms an, welches den gesamten Bildschirminhalt ausliest und nach E000 schreibt:

```
1000    LDY #00
1002    LDA 0400,Y
1005    STA E000,Y
1008    LDA 0500,Y
100B    STA E100,Y
100E    LDA 0600,Y
1011    STA E200,Y
1014    LDA 0700,Y
1017    STA E300,Y
101A    DEY
101B    BNE 1002
...
```

Wie Sie sehen, erfordert das durch die Tatsache, daß vier Blöcke zu je 256 Bytes übertragen werden müssen, immerhin schon 28 Bytes Programmtext. Nun soll die indirekt-indizierte Adressierung verwendet werden, um dieselbe Aufgabe zu lösen. Wir legen zunächst zwei Zeiger auf der Zeropage fest:
FA/FB sollen die Bildschirmadresse enthalten

FC/FD die Zieladresse ab E000.

```
1000    LDA #00
1002    STA FA
1004    STA FC
1006    LDA #04
1008    STA FB
100A    LDA #E0
100C    STA FD
```

Damit sind die Zeiger festgelegt. Es sind vier Blöcke zu je 256 Bytes zu übertragen. Diese Blockanzahl legen wir ins X-Register als Zähler:

```
100E    LDX #04
```

Dann laden wir ins Y-Register ebenfalls einen Zähler (den Index):

```
1010    LDY #00
```

Jetzt kann die eigentliche Übertragungsschleife starten:

```
1012    LDA (FA),Y
1014    STA (FC),Y
1016    DEY
1017    BNE 1012
```

Wenn das Y-Register wieder bei 0 angekommen ist (von der ersten 0 nach einem Unterlauf — siehe dazu Folge 3 — über FF, FE und so weiter bis 0), ist der erste Block übertragen. Wir erhöhen nun das MSB beider Zeiger um 1:

```
1019    INC FB
101B    INC FD
```

Außerdem zählen wir den Blockzähler um 1 herunter:

```
101D    DEX
101E    BNE 1012
...
```

Wenn das Programm auf diese Weise auch drei Bytes mehr Speicherplatz braucht, ist doch leicht der Vorteil zu sehen: Müssen wir nämlich (statt nur vier) mehr Blöcke übertragen (bis zu 255), dann verändert sich unser zweites Programm um keinen Deut (außer dem Zähler im X-Register, der nun mit der jeweils anderen Block-Anzahl geladen wird), während die erste Programmtechnik für jeden weiteren Block um sechs Bytes erweitert werden muß.

Es gibt noch eine ganze Reihe von Anwendungsmöglichkeiten, die die indirekt-indizierte Adressierung so attraktiv machen. Für Geschwindigkeitsfanatiker (ich selbst bin bei Grafik-Fragen auch einer!) muß aber gesagt werden, daß dem Speicherplatzvorteil ein Geschwindigkeitsnachteil gegenübersteht. Jeder indirekt-indiziert adressierte Befehl braucht einen Taktzyklus länger als der vergleichbare absolut-indizierte Befehl. Zu diesen Feinheiten werden wir aber in späteren Folgen noch kommen.

Die indiziert-indirekte Adressierung

Wenden wir uns nun der letzten noch fehlenden Adressierungsart zu, der indiziert-indirekten. Man nennt sie auch »vorindizierte indirekte« oder »indirekt X« Adressierung. Sehen wir auch hier zunächst ein Beispiel an:

STA (FA,X)

Auch hier drückt die Klammer wieder aus, daß der Klammerinhalt ein Zeiger ist. Das ist jetzt aber nicht das Bytepaar FA/FB, sondern zur angegebenen Adresse FA soll noch der Inhalt des X-Registers addiert werden. Nehmen wir mal an, dort stünde eine 2, dann wird der Zeiger FC/FD mit diesem Befehl angesprochen, denn FA+2=FC und entsprechend FB+2=FD. Wenn in den Speicherstellen FA bis FF folgender Inhalt zu finden ist:

```
00FA  00
00FB  04  FA/FB = 0400
00FC  00
00FD  E0  FC/FD = E000
00FE  10
00FF  80  FE/FF = 8010
```

dann könnte das eine ganze Tabelle von Zeigern sein, die jeweils durch den X-Registerinhalt angesprochen werden. Der Akkuinhalt wird in unserem Beispiel nach 0400 geschrieben, wenn im X-Register 0 steht, nach E000, wenn das X-Register eine 1 enthält und nach 8010, wenn statt dessen eine 2 im X-Register zu finden ist.

Sie werden sich vielleicht auch bei diesem Beispiel gefragt haben, was passiert, wenn im X-Register unseres Beispiels eine 3 steht. Nun, unser 8-Bit-Prozessor läuft über, und wir finden einen Zeiger 00/01.

Rein theoretisch ist diese Adressierungsweise ganz interessant. Aber auf der Zeropage ist's reichlich eng, und nur selten kommt man daher in die Lage, dort eine Zeigertabelle einzurichten, die man mittels des X-Registerinhalts und der indiziert-indirekten Adressierung abgreifen kann. Die Bedeutung dieser Adressierungsart ist also

Speicherstelle 1 Bits 2	1	0	$A000-$BFFF	$D000-$DFFF	$E000-$FFFF
1	1	1	Basic	I/O	Kernal
1	1	0	RAM	I/O	Kernal
1	0	1	RAM	I/O	RAM
1	0	0	RAM	RAM	RAM
0	1	1	Basic	Zeichen	Kernal
0	1	0	RAM	Zeichen	Kernal
0	0	1	RAM	Zeichen	RAM
0	0	0	RAM	RAM	RAM

Tabelle 1 zeigt, welche Bausteine bei verschiedener Belegung der Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind. (Frei nach Dr. Hauck, 64'er Ausgabe 11/84, Seite 136)

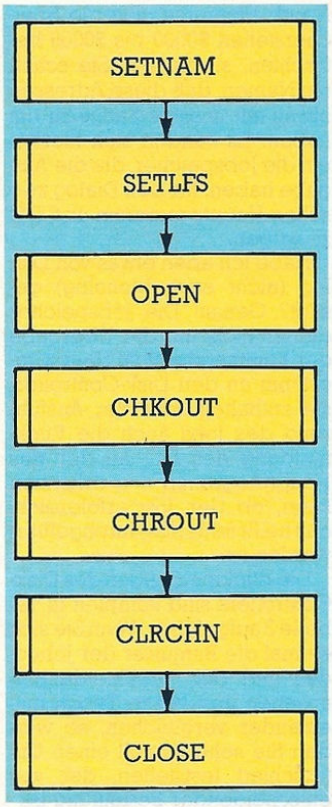

Bild 3. Die Abfolge der Routineaufrufe

nitors erforderlich macht, solange man sich noch nicht sicher ist, ob Fehlermeldungen auftauchen). Die Null im Akku besagt, daß kein Filename gewünscht ist. Dann kommt SETLFS:

```
6023   LDA  #04
6025   LDX  #04
6027   LDY  #FF
6029   JSR  FFBA
602C   BCS  6053
```

Es wurde ein File festgelegt mit der logischen Filenummer 4, der Geräteadresse 4 und ohne Sekundäradresse. Jetzt geben wir das OPEN-Kommando:

```
602E   JSR  FFC0
6031   BCS  6053
```

Der Ausgabekanal wird definiert mit CHKOUT:

```
6033   LDX  #04
6035   JSR  FFC9
6038   BCS  6053
```

Damit sind alle Vorbereitungen erledigt, und die Zeichenausgabe kann wie im ersten Programm durchgeführt werden mit CHROUT:

```
603A   LDY  #00
603C   LDA  6000,Y
603F   BEQ  604A
6041   JSR  FFD2
6044   BCS  6053
6046   INY
6047   JMP  603C
```

Alle Zeichen sind nun ausgedruckt. Wir rufen CLRCHN auf:

```
604A   JSR
       FFCC
```

Als letzte Routine folgt nun noch CLOSE:

```
604D   LDA  #04
604F   JSR  FFC3
6052   RTS
```

Damit wurde das File Nummer 4 geschlossen. Anschließend erfolgte der Rücksprung aus dem Programm. Für die Fehlerbehandlung habe ich nur einen BRK vorgesehen, der sofortigen Registerüberblick erlaubt, wenn zum Beispiel der SMON im Speicher enthalten ist.

```
6053   BRK
```

Ohne Monitor im Speicher kann der Computer allerdings abstürzen oder im besten Fall einen Basic-Warmstart durchführen. Wenn Sie sowas also für Ihre Zwecke programmieren möchten, sollten Sie einen anderen Weg suchen, die Fehler aufzufangen. Man hat ja nicht immer einen Monitor eingeladen.

Mit diesen sieben Kernal-Routinen soll's für diesmal genug sein. In der Dezember-Ausgabe des 64'er haben B. Schneider und K. Schramm in ihrer Serie »In die Geheimnisse der Floppy eingetaucht« gezeigt, wie man mittels der besprochenen Routinen, und einiger anderer, auch die Diskettenstation ansprechen oder sogar Floppy und Drucker zum »Spooling« veranlassen kann. Das habe ich zwar schon öfter gesagt, muß es aber trotzdem immer wieder tun: Durch das Nachvollziehen fremder Programme kann man sehr viel lernen.

Oje, mein Versprechen, diesmal mit den Fließkommazahlen weiterzumachen, kann ich nicht halten. Auch unser Programmprojekt kommt nicht mehr dran. Beides hätte den Umfang dieser Folge mit Sicherheit gesprengt. Ich gebe Ihnen aber mein großes Ehrenwort, daß wir in der nächsten Ausgabe beide Themen weiterbehandeln.

(Heimo Ponnath/gk)