Von Basic zu Assembler (Teil 5)

Viele Fragen tauchten während dieses Kurses auf. An dieser Stelle sollen Sie Antworten auf die drängendsten Fragen erhalten, die zum Verständnis des Kurses beitragen sollen.

Viele Leser haben uns zu ihren Problemen mit Assembler geschrieben. Einige Fragen sind so häufig gestellt worden, daß sie in dieser Folge unseres Kurses beantwortet werden sollen.

Stellvertretend für viele Leser wünscht sich Herr R. Benkovic eine leichtverständliche Einführung in die Maschinensprache.

Außer Ihnen fragen eine ganze Menge der Leser nach einer leichtverständlichen Einführung in die Maschinensprache. Der Kurs »Von Basic zu Assembler« baut auf schon vorhandenen grundlegenden Kenntnissen der Maschinensprache auf, die in der Serie »Assembler ist keine Alchemie« (64'er-Hefte 9/84 bis 10/85) vermittelt wurden. Er ist also keine Einführung in die Maschinensprache, sondern soll ein Umdenken von Basic zur Maschinensprache durch Vorstellung von verschiedenen Techniken anhand kleiner Beispiele fördern. Von vielen Lesern wurde mir mitgeteilt, daß sie den Kurs »Assembler ist keine Alchemie« gut zum Einstieg in die Maschinensprache verwenden konnten. Zwar zieht das Niveau in den letzten Folgen etwas an, aber die Voraussetzungen — besonders in den ersten Teilen — sind so gering gehalten, daß man auch als völliger Neuling in diese Sprache eingeführt wird. Weil sich dieser Kurs doch über eine ganze Reihe von Folgen erstreckte, wurde er noch einmal komplett abgedruckt, und zwar im 64'er-Sonderheft 8/85. Ich rate Ihnen deshalb, am besten zuerst einmal mit dem Durcharbeiten dieses Kurses zu beginnen. Nach und nach werden Sie dann auch die Reihe »Von Basic zu Assembler« besser verstehen und als Training benutzen können.

Auch Herr C. Scheper möchte Assembler-Grundlagen erklärt haben. Außerdem aber fragt er — wie viele andere Leser —, wie man Assembler-Listings eingeben könne.

Ein Assembler-Listing einzugeben ist — jedenfalls beim C 64 — nur möglich mit Hilfe eines Programmes, das man »Assembler« nennt. Darunter versteht man (die Namensgleichheit ist etwas unglücklich) ein Programm, das mit Hilfe des sogenannten »Editors« die Erstellung von Programmtext in der Sprache Assembler erlaubt, die Verwendung von Kommentaren, von Sprungmarken und anderes mehr. Weiterhin aber kann man damit den auf diese Weise geschriebenen »Quelltext« (das ist meistens das, was Sie in den Zeitschriften abgedruckt finden) auch übersetzen in die eigentliche Maschinensprache, den sogenannten »Objektcode«, und diesen dabei an einer bestimmten Stelle im Speicher oder auch auf der Diskette ablegen. Solche Assembler-Programme kann man kaufen. Ich verwende im Kurs den »Hypra-Ass«, der komplett im Assembler-Sonderheft (Ausgabe 8/85) abgedruckt wurde und ein wirklich guter Assembler ist. Wie man den benutzt, wird dort genau beschrieben, sowie auch in den Ausgaben 12/85 und 1/86 des 64'er-Magazins.

Man geht also so vor: Hypra-Ass laden und durch RUN starten. Nun können Sie den abgedruckten Quelltext eintippen, dann abspeichern oder ausdrucken. Durch ein weiteres RUN starten Sie die Assemblierung, die diesen Programmtext in ein direkt ausführbares Maschinenprogramm im Speicher verwandelt, den sogenannten Objektcode. Ich empfehle Ihnen, vor dem Hypra-Ass noch einen Monitor — wie den SMON — beispielsweise nach $C000 zu laden (das geht mittels des absoluten Ladebefehls, bei dem man nach der Gerätenummer 8 (oder 1 beim Kassettenbetrieb) noch eine 1 angibt:

```
LOAD"SMON $C000",8,1
```

Auf diese Weise hat man eine komplette Programmierausstattung im Computer: Den Assembler und Editor von Hypra-Ass und den SMON, mit dem man dann auch Speicherbereiche disassemblieren kann, Umrechnungen werden erleichtert, man kann auch Objektcode direkt laden oder abspeichern und so fort. Auch den SMON finden Sie im schon erwähnten Assembler-Sonderheft.

Einige Leser haben Probleme mit Adressenangaben im Computerformat. Was ist ein LSB und ein MSB, was ist das Hi- und das Low-Byte einer Adresse?

Bei jeder Zahl mit mehreren Stellen, beispielsweise der Dezimalzahl 6001, gibt es Stellen mit großer Bedeutung (hier die 6) und solche mit geringerer Wichtigkeit (hier die 1). Es macht eben einen großen Unterschied, ob man für einen Gebrauchtwagen statt 6001 Mark nun 7001 Mark bezahlen soll, wohingegen es uns kaum kümmert, ob wir 6001 Mark oder 6002 Mark dafür loswerden. Dasselbe gilt natürlich auch, wenn man statt mit Dezimalzahlen mit Hexadezimalzahlen arbeitet.

Die Hex-Zahlen zwischen $0000 und $FFFF (dezimal 0 bis 65535) sind besonders interessant im Zusammenhang mit 8-Bit-Computern, denn damit ist genau der durch den Zentralprozessor erreichbare Adreßraum erfaßbar. Stellt man die höchste Adresse $FFFF in der Form dar, wie sie der Prozessor erkennt, also als Binärzahl, dann findet man:

11111111 11111111

Das ist aber eine Zahl mit 16 Bit. Ein 8-Bit-Computer kann solch eine Adresse also nur in 2 Häppchen verdauen, von denen jedes 8 Bit lang ist. Stellt man sich jede Hex-Zahl rechtsbündig mit 4 Stellen und vorlaufenden Nullen dar (also statt $123 nun $0123 oder statt $FE nun $00FE), dann ergibt die Aufspaltung dieser Zahl zwischen der zweiten und der dritten Stelle zwei 8-Bit-Zahlen. Beispielsweise kann man dann die Zahl $CFFE auftrennen in $CF und $FE. Erinnert man sich an die oben erwähnte Wichtigkeit der einzelnen Stellen und an die Tatsache, daß eine 8-Bit-Zahl auch ein »Byte« genannt wird, dann existiert hier also ein bedeutendes Byte (nämlich im Beispiel $CF) und ein weniger wichtiges (das ist hier $FE). Das erstere wird mit der Bezeichnung MSB (das kommt von »Most Significant Byte« = bedeutsamstes Byte oder auch häufig »High-Byte« oder auch »Hi-Byte«) versehen. Das andere nennt man LSB (das heißt »Least Significant Byte« = am wenigsten bedeutsames Byte oder manchmal auch »Low-Byte«). Die Zahl $CFFE ist also auftrennbar in ein MSB ($CF) und ein LSB ($FE).

Im allgemeinen bekommt man von dieser Auftrennung nur relativ selten etwas zu sehen, und es ist ebenso falsch, eine Zahl wie $CFFE plötzlich mal als $FECF zu schreiben, wie es auch bei der Dezimalzahl 6001 den Autohändler verärgern würde, einen Preis von 0160 für den Gebrauchtwagen zu bekommen. Alle handelsüblichen Assemblerprogramme beispielsweise verlangen die normale Adresse. Wenn Sie also etwas an die erste Position des Bildschirmes schreiben wollen, dann machen Sie das mit der Befehlszeile:

```
STA $0400
```

Die Hex-Zahl $0400 entspricht im Dezimalsystem der Zahl 1024 und ist die Adresse der ersten Bildschirmspeicherzelle. Eine andere Sache ist es, wie solche Adressenangaben im Speicher unseres Computers aufbewahrt werden. Der oben genannte Befehl ist ein 3-Byte-Befehl: Ein Byte ist reserviert für den Code des Befehles STA, zwei Byte braucht die Adresse $0400, wie Sie vorhin sehen konnten. Assembliert man das Programm, das diese Befehlszeile enthält und blickt mit einem Monitor in die Speicherstellen, die nun den Objektcode (also den vom Mikroprozessor erkennbaren Befehlscode) enthalten, dann findet man anstelle dieser Zeile nun:

```
... 8d  00  04 ...
```

8D ist der Befehlscode für das absolute STA. Dann folgt 00, wobei es sich um das LSB der Adresse 0400 handelt und schließlich das MSB 04. Intern bewahrt der Computer also seine 16-Bit-Adressen in der Reihenfolge LSB-MSB auf.

Der Programmierer wird mit dieser Art der Speicherung von Adressen immer dann konfrontiert, wenn er eine indirekte Adressierung verwendet und Vektoren mit einer Adresse belegen muß. Weitere Einzelheiten dazu aber lesen Sie bitte im Kurs »Assembler ist keine Alchemie«

nach (vollständig abgedruckt im Sonderheft 8/85 des 64'er-Magazins). Dort sind in den Kapiteln 28 und 36 die drei Arten der indirekten Adressierung erklärt.

A. Auer fragt: Wie kann ich die unten stehenden Basic-Zeilen in Maschinenprogramme umwandeln, daß sie durch SYS... aufgerufen werden können?

```
100 POKE 646,9
```

oder

```
100 A = 9:POKE A,9
```

Dem POKE-Kommando in Basic entsprechen die Befehle STA, STX und STY in Assembler, die jeweils den Inhalt des Akkumulators, des X- oder Y-Registers irgendwo im Speicher ablegen. Wo der Inhalt abzulegen ist, das erfährt der Mikroprozessor aus der nach dem Befehl angegebenen Adresse. Die Angabe der Adresse wiederum geschieht meist in Hexadezimalzahlen (viele Assembler-Programme verstehen auch Dezimaladressen oder akzeptieren vorher festgelegte Kennworte, wie beispielsweise der Assembler Hypra-Ass). Der Adresse 646 beispielsweise entspricht im Hexadezimalsystem die Angabe $0286. Außerdem muß natürlich zuvor auf irgendeine Weise in das angesprochene Register der abzulegende Wert gepackt werden. Dem Basic-Befehl POKE 646,9 entspricht daher die Assembler-Sequenz:

```
LDA #$09
STA $0286
```

Anstelle von LDA und STA kann auch verwendet werden: LDX und STX oder LDY und STY. Damit diese Sequenz von Basic aus mittels des SYS-Befehls aufrufbar ist, muß das kleine Programm einen Befehl enthalten, der nach der Abarbeitung wieder ins Basic zurückführt (das ist RTS) und es muß assembliert im Speicher stehen (Hypra-Ass beginnt nach der Fertigstellung des Programmtextes und einer Angabe der gewünschten Startadresse das Assemblieren, wenn man RUN <RETURN> eingibt). Bei einer Startadresse von $C000 sähe unser komplettes Programm dann so aus:

```
C000    LDA #$09
C002    STA $0286
C005    RTS
```

Es wird dann durch den Basic-Befehl SYS 49152 (das ist der Dezimalwert von $C000) gestartet.

Etwas schwieriger wird es, wenn die Zieladresse variabel gehalten werden soll, wie es in der Basic-Befehlszeile

```
A = 1024 : POKE A,9
```

angedeutet ist. Es gibt dazu mehrere Möglichkeiten der Assembler-Programmierung. Die flexibelste Lösung bietet sicherlich die Verwendung der indirekt-indizierten Adressierung. An die Stelle der Basic-Variablen A tritt hier ein sogenannter Vektor. Das sind zwei aufeinanderfolgende Speicherstellen in der Zeropage (beispielsweise $FA und $FB), in die man die Zieladresse in der Form LSB/MSB einträgt. Die Adresse 1024 lautet im Hex-Format $0400. Ihr LSB ist $00, das MSB besteht aus $04. In die Speicherstelle $FA muß daher $00, in $FB muß $04 geschrieben werden, damit nun unser Vektor auf die erste Bildschirmspeicherstelle zeigt:

```
LDA #$00
STA $FA
LDA #$04
STA $FB
```

Außerdem muß das Y-Register noch auf Null gesetzt werden, denn der Befehl, der diese Art der Adressierung verwendet, lautet STA ($FA),Y. In der Klammer wird die erste Vektoradresse genannt. Der Akkuinhalt landet in der Speicherstelle, auf die dieser Vektor zeigt (also $0400), plus dem Inhalt des Y-Registers. Wäre das Y-Register beispielsweise gleich 1, dann würde der Akku-Inhalt nach $0401 geschrieben. Der Fortgang des Programmes muß daher lauten:

```
LDY #$00
LDA #$09
STA ($FA),Y
RTS
```

Diese Art der Programmierung wird häufig in Schleifen angewendet, wo man dann Y verändert, um über 255 Speicherstellen verfügen zu können. Außerdem kann der Inhalt von $FA/FB verändert werden. So hat man Zugriff auf den gesamten Speicher.

Stellvertretend für eine ganze Reihe anderer Leser schreibt Herr M. Goldberg:

Mein Problem liegt in den verschiedenen Adressierungsarten. Die unmittelbare Adressierung macht mir noch keine Probleme, doch die absolute Adressierung und die anderen Adressierungsarten machen mir sehr zu schaffen.

Im Rahmen einer kürzeren Abhandlung ist Ihre Frage nach den Adressierungsarten nur recht knapp zu beantworten. Wenn Sie also nach diesen Anworten noch Probleme haben sollten, dann empfehle ich Ihnen die Lektüre des Kurses »Assembler ist keine Alchemie« (vollständig abgedruckt im Sonderheft 8/85 des 64'er-Magazins).

13 Arten der Adressierung gibt es im 6502-Assembler (der ja auch den 6510 des C 64 und den 8502 des C 128 programmiert), von denen ich im nachfolgenden aber nur einige häufiger benutzte vollständig erklären werde (siehe auch Tabelle 1):

Implizit

Das ist eine Form der Adressierung, die nur bei bestimmten Befehlen möglich ist, nämlich bei solchen, in denen die Adressierung praktisch schon enthalten ist (das heißt ja implizit). Beispielsweise besagt der Befehl TAY, daß der Akkuinhalt ins Y-Register kopiert werden soll.

Akkumulator

Wenn beispielsweise hinter dem Befehl ROL keine Adresse oder nur ein A steht, dann heißt das, daß der Inhalt des Akkumulators nach links rotiert werden soll. Man spricht dann von der Akkumulator-Adressierung.

Absolut

Die absolute Adressierung spricht einzelne Orte im Speicher direkt an. Man unterscheidet zwei Arten dieser absoluten Adressierung, nämlich diejenige, die sich auf 16-Bit-Adressen bezieht (wie LDA $CEF2) und diejenige, die sich auf Adressen aus der Zeropage in 8-Bit-Format bezieht (wie beispielsweise STA $FA).

Unmittelbar

Einige Befehle erlauben das unmittelbare Eintragen von Werten in Register. Soll zum Beispiel der Wert 255 (das ist in Hex-Zahlen ausgedrückt $FF) in den Akku geladen werden, dann verwendet man die unmittelbare Adressierung: LDA #$FF. Das Kennzeichen dieser Adressierung ist das vorangestellte Doppelkreuz (#).

Indiziert

Nun wird's etwas komplizierter. Es gibt vier Arten der reinen indizierten Adressierung. Bei allen vieren findet man nach dem Befehlswort eine Adresse, dann ein Komma und danach entweder ein X oder ein Y. Als Beispiel kann man schreiben:

```
LDA $C000,Y
```

Gehen wir erst mal von der absoluten Adressierung aus, also von LDA $C000. Das bedeutet ja, daß der Inhalt der Speicherzelle $C000 in den Akku geladen wird. Ist der Inhalt des Y-Registers gleich Null, dann erfüllt LDA $C000,Y genau denselben Zweck. Der Inhalt des jeweiligen Indexregisters (also X oder Y) wird nämlich zur genannten Adresse hinzugezählt. LDA $C000,Y lädt daher den Inhalt der Speicherstelle $C005, wenn sich im Y-Register der Wert 5 befindet. Die vier verschiedenen reinen indizierten Adressierungsarten (jeweils mit Beispielen) sind:

```
LDA $C000,Y     Y-indiziert
STA $C0FE,X     X-indiziert
CMP $2F,X       Zeropage-X-
                indiziert
LDX $A1,Y       Zeropage-Y-
                indiziert
```

Indirekt

Man unterscheidet drei Arten der indirekten Adressierung, die alle durch das Prinzip des toten Briefkastens erklärbar sind. Wenn Sie Leser von Agenten-Thrillern sind, dann kennen Sie dieses Prinzip: Ein Agent soll einen Kontaktmann treffen. Er weiß aber nicht, an welchem Ort dieses Treffen stattfinden wird. Statt dessen aber existiert in einem Park der Stadt ein hohler Baum, in dem zu bestimmten Zeitpunkten ein Zettel mit der Adresse des Treffpunktes liegt. Sowas nennt man einen toten Briefkasten. Die reine indirekte Adressierung existiert im 6502-Assembler nur für den JMP-Befehl. Dem toten Briefkasten entspricht hier ein sogenannter Vektor (oder auch Zeiger). Das sind zwei aufeinanderfolgende Speicherstellen, in denen sich in der LSB/MSB-Form die eigentliche Adresse befindet. So bedeutet der Befehl JMP ($4000), daß der Programmzähler quasi in den toten Briefkasten $4000/$4001 sehen muß, um darin das Sprungziel zu finden. Soll der Sprung beispielsweise zu einem Programm stattfinden, das bei $CEFD beginnt, dann muß sich das LSB (also $FD) in der Speicherstelle $4000, das MSB (also $CE) in $4001 befinden, damit JMP ($4000) diesen Sprung richtig ausführt.

Wesentlich häufiger wird die indirekt-indizierte Adressierung gebraucht. Das sieht dann zum Beispiel so aus:

```
LDA ($FA),Y
```

$FA/FB ist auch hier wieder der tote Briefkasten. Zu der darin befindlichen Adresse wird aber noch der Inhalt des Y-Registers addiert. Als toter Briefkasten können hier übrigens nur Zeropage-Adressen dienen. Nehmen wir mal an, im eben genannten Beispiel zeige der Vektor $FA/FB (also der tote Briefkasten) auf die Adresse $E000 und im Y-Register befände sich der Wert $A1. Dann lädt der Akku den Inhalt der Speicherstelle $E0A1.

Höchst selten verwendet man die indiziert-indirekte Adressierung. Sie wird im Beispiel STA ($FA,X) gebraucht. Der tote Briefkasten ist hier nicht mehr $FA/FB, sondern $(FA + X) / (FB + X).

Relativ

Mit dieser Adressierungsart werden Sie als Programmierer kaum zusammenstoßen. Sie wird intern bei den Branch-Befehlen (wie BCC und so weiter) verwendet. Ich habe aber schon seit Jahren keinen Assembler mehr zu Gesicht bekommen, der die Angabe einer relativen Adresse verlangt. Alle sind mit der absoluten Angabe eines Verzweigungszieles zufrieden.

Relativ heißt in diesem Zusammenhang, daß angegeben wird, wieviele Bytes relativ zum derzeitigen Programmzählerstand vorwärts oder rückwärts gesprungen werden muß. Man spürt die Eigenart dieser Adressierung nur dann, wenn man mal einen besonders weiten Sprung durch einen Branch-Befehl ausführen lassen möchte: Es ist nämlich hier nur möglich, maximal 127 Byte vorwärts oder 128 Byte rückwärts zu verzweigen.

H. Metschulat stellt zwei Fragen:
1) Wie kann man beim SYS-Befehl Parameter übergeben und wo werden sie gespeichert? Beispielsweise

```
SYS 49152,"64er"
```

2) Was bedeutet der USR(X)-Befehl? Ich weiß, daß er etwas mit Maschinensprache zu tun hat, aber was?

Die Übergabe von Parametern durch die oft praktizierte Form »SYS Adresse,a,b,...« ist beim C 64 im Gegensatz zum C 128 durch Basic nicht unterstützt. Durch den Aufruf »SYS Adresse« geht die Kontrolle über das weitere Geschehen voll auf das Maschinenprogramm über, das bei »Adresse« beginnt. Nun hat man in diesem Maschinenprogramm dafür zu sorgen, daß die hinter »Adresse« stehenden Parameter gelesen, in geeigneter Form gespeichert und verarbeitet werden. Die Ergebnisse müssen weiterhin auf irgendeine Weise ausgegeben oder an das Basic-Programm überreicht werden. Zu guter Letzt soll das Basic-Programm an der richtigen Stelle weiterlaufen. Wie all das realisiert werden kann, ist an dieser Stelle nicht darstellbar. Ich bitte Sie um Geduld, denn das wird auch Thema einer späteren Folge des Kurses »Von Basic zu Assembler« sein. Weiterhin gebe ich Ihnen eine Literatur-Empfehlung: Das Buch von W. Kassera und F. Kassera, »C 64 Programmieren in Maschinensprache«, erschienen im Verlag Markt & Technik als MT 830 (ISBN 3-89090-168-9), kostet inklusive einer Diskette 52 Mark und widmet sein Kapitel 10 dieser Frage.

Auch der Basic-Befehl USR ruft ein Maschinenprogramm auf. Das Argument dieses Befehls übergibt aber nicht wie bei SYS die Startadresse, sondern einen Parameter. Die Startadresse des angesteuerten Programmes wird hier in einem Vektor angegeben, dem USR-Vektor bei 785/786 (oder $311/$312). Im Einschaltzustand zeigt dieser Vektor in eine Routine zur Ausgabe eines SYNTAX ERROR. Hat man daher versäumt, dem Vektor vor dem USR-Aufruf einen neuen Wert zuzuweisen, dann bricht das Programm mit dieser Fehlermeldung ab. Die Startadresse des eigenen Maschinenprogrammes muß in der Form LSB/MSB (beispielsweise durch POKE-Kommandos) im USR-Vektor vorhanden sein. Der Aufruf geschieht zum Beispiel in der Form X = USR(Y). Möglich sind aber auch andere Verwendungen, wie PRINT USR(A) oder B = SIN(C)*USR(A) etc. Das Argument befindet sich nach dem Sprung ins Maschinenprogramm im FAC (dem Fließkomma-Akkumulator1) als Fließkommazahl (im FLPT-Format). Nun liegt wieder die weitere Verantwortung beim Programmierer. Wenn er mit diesem Argument Berechnungen anstellen will (man kann auch Dummy-Werte verwenden, die dann nicht weiter verwendet werden), muß er nun für die richtige Handhabung sorgen. Die Kenntnis von Interpreter-Routinen erleichtert diese Aufgabe sehr. Auch hierzu möchte ich Sie auf weitere Folgen des Kurses vertrösten und auf das vorhin schon erwähnte Buch verweisen. Auch im Kurs »Assembler ist keine Alchemie« (vollständig abgedruckt im Sonderheft 8/85 des 64'er-Magazins) wird auf den USR-Befehl eingegangen. Wenn aus dem Maschinenprogramm ein in Basic weiterzuverarbeitendes Ergebnis übergeben werden soll, dann muß man dafür sorgen, daß es als Fließkommazahl im FAC steht, bevor man ins Basic-Programm zurückkehrt. Das Ergebnis erfährt dann dieselbe Behandlung wie das einer beliebigen anderen Basic-Funktion (beispielsweise SIN(X)). Fand der USR-Aufruf also durch PRINT USR(X) statt, dann steht das Ergebnis hinterher auf dem Bildschirm.

Einige — schon etwas fortgeschrittene — Leser stellen Fragen wie Herr R. Lersch:
1) Was kann man mit dem Fließkomma-Akkumulator machen?
2) Was kann man mit dem FAC machen?
3) Was kann man mit dem ARG machen?

Ich gehe davon aus, daß Ihnen der Begriff der Fließkommazahl bekannt ist, weshalb ich darauf auch nur ganz kurz eingehe. In Basic unterscheidet man die Integer-Zahlen, also ganze vorzeichenbehaftete Zahlen, und die Fließkommazahlen (die auch Gleitkommazahlen, Zahlen in wissenschaftlicher Darstellung oder Reals genannt werden). Je nach Größe der Integer-Zahl benötigt man dann 1 oder 2 Byte (das LSB und das MSB), um sie abzuspeichern. Es gibt auch Möglichkeiten, andere Formate selbst zu verwalten. Fließkommazahlen bestehen aus mehreren Teilen: Mantisse, Exponent und Vorzeichen, wobei auch der Exponent vorzeichenbehaftet ist. Eine solche Fließkommazahl ist beispielsweise:
-1985,123 * 10^+5

Im Beispiel ist eine Fließkommazahl im Dezimalsystem gezeigt. Der Computer verwendet natürlich das binäre Zahlensystem. Falls Ihnen weder Zahlensysteme wie das Hexadezimal- und das Binärsystem noch Fließkommazahlen vertraut sind, dann gebe ich Ihnen noch einige Literaturhinweise: Zum einen hilft Ihnen der Kurs »Assembler ist keine Alchemie«, der vollständig abgedruckt im Sonderheft 8/85 des 64'er-Magazins vorliegt und zum anderen (besonders bei den Zahlensystemen) mein Buch »C 64 Wunderland der Grafik«, das im Markt & Technik Verlag unter der Nummer MT 90363 erschienen ist. Außerdem plane ich für eine der nächsten Folgen des Kurses »Von Basic zu Assembler« einen Beitrag, der sich dieser Fragestellung annimmt.

Fließkommazahlen speichert der C 64 in zwei verschiedenen Formaten: Im MFLPT-Format (»Memory floating point«) in 5 Byte und im FLPT-Format (»Floating point«) in 6 Byte. Letzteres Format trifft man in den sogenannten Fließkomma-Akkumulatoren an. Davon gibt es deren zwei in unserem Computer, die FAC und ARG genannt werden (Manchmal liest man auch »Fließkomma-Akkumulator 1« und »2« dafür). Der FAC belegt beim C 64 die Speicherstellen 97 bis 102 ($61 bis $66), der ARG die von 105 bis 110 ($69 bis $6E). Die Belegung ist dabei folgende:

FAC	ARG	
97	105	Exponent inklusive Exponentenvorzeichen
98	106	
99	107	Mantisse
100	108	in vier Bytes
101	109	
102	110	Vorzeichen der Mantisse

Was der normale Akkumulator bei 1-Byte-Operationen ist, das leistet der FAC bei Operationen mit Fließkommazahlen. Über seine Bedeutung gewinnt man noch mehr Klarheit, wenn man sich die Tatsache ins Gedächtnis holt, daß nahezu alle Operationen mit Zahlen beim C 64 im Fließkommaformat durchgeführt werden. Wenn man beispielsweise den Sinus einer Zahl berechnen möchte, dann packt man das Argument in den FAC, ruft dann die Interpreter-Routine SIN auf und findet das Ergebnis wieder im FAC. Auf diese Weise kann man eine ganze Menge mathematischer Operationen mit der eingebauten Firmware erledigen. Der ARG kann oft verwendet werden bei Operationen mit zwei Zahlen (wie beispielsweise bei der Multiplikation). Die Handhabung der FLPT- und MFLPT-Formate ist — wie Sie sich sicher vorstellen können — nicht gerade einfach. Deshalb tut man als Assembler-Programmierer gut daran, dazu weitgehend Interpreter-Routinen zu gebrauchen, wenn es nicht gerade darum geht, Geschwindigkeitsrekorde zu brechen und deshalb eigene, schnellere Wege zu verfolgen, als die manchmal etwas umständlichen Serpentinen der Commodore-Firmware-Schöpfer. Auch hier muß ich Sie um etwas Geduld bitten: Interpreter-Routinen werden ebenfalls unser Thema im Kurs »Von Basic zu Assembler« sein. Sollten Sie aber nicht warten wollen, dann schauen Sie mal in das Buch von W. Kassera und F. Kassera, »C 64 Programmieren in Assembler«, erschienen im Markt & Technik Verlag unter der Nummer MT 830. Eine Tatsache wird bei einigen dieser sieben Fragen wohl dem Leser deutlich: Oft sind es gerade die einfach scheinenden Fragen, die eine komplizierte Antwort erfordern. Manche Antworten führen sofort zu neuen Fragen: Eine Adresse in 2 Bytes aufzutrennen, ist nach der Antwort schon klar. Was aber ist eine Hex-Zahl und warum ist 1024 dasselbe wie $0400? In der nächsten Folge werden wir zuerst die Schleifenprogrammierung weiter behandeln mit einer selbstmodifizierenden Programmtechnik, uns dann aber den Zahlensystemen zuwenden und den Fragen danach, wie man sie ineinander umrechnen kann und wozu wir uns überhaupt mit ihnen auseinandersetzen müssen.

(Heimo Ponnath/dm)

Adressierung	Bytes	Beispiel
Implizit	1	TAX
Absolut	3	LDA $C0A8
Unmittelbar	2	LDA #$4F
Indiziert	3	LDA$9000,X
Indirekt Indiziert	2	LDA($02),Y
Indiziert Indirekt	2	STA($01,X)
Indirekt	3	JMP($,4301)
Zeropage	2	LDA$80
Relativ	2	BEQ$03

Tabelle 1. Hier noch einmal die verschiedenen Arten der Adressierung