DAS ELEKTRONISCHE MAGAZIN

INPUT 64

SPECIAL 2

Assembler und Editor:

INPUT-ASS

Für C64 und C128

Der komplette Kurs:

6502-Maschinensprache

Lernen direkt am Rechner

Maschinensprache-Monitor

Für Rechner und Floppy

6502-Simulator

Debugging im Source-Code

Re-Assembler

Vom Hexcode zum Quelltext

Über 200 KByte Software auf doppelseitig bespielter Diskette

Unverbindliche Preisempfehlung

ÖS 220,– / sfr 29,80

DM 29,80

Das Magazin für C64- und C128-Anwender

Hinweise zur Bedienung

Dieses INPUT 64-Special ist nicht nur einfach eine Programmsammlung auf Diskette, sondern ein Elektronisches Magazin. Es enthält ein eigenes Betriebssystem mit Schnellader und komfortabler Programmauswahl. Die Bedienung ist kinderleicht:

Bitte entfernen Sie vor dem Laden eventuell vorhandende Steckmodule, und schalten Sie den Rechner einmal kurz aus und wieder ein. Geben Sie nun zum Laden der Diskette

LOAD "INPUT*",8,1 und RETURN

ein. Alles Weitere geschieht von selbst.

Es wird nun zunächst ein Schnellader initialisiert. Besitzen Sie ein exotisches Laufwerk oder ist Ihre Floppy bereits mit einem hardwaremäßigen Beschleuniger ausgerüstet, kann es zu Konflikten mit unserem SuperDisk kommen. In diesem Falle sollten Sie versuchen, die Diskette mit

LOAD "LADER*",8,1 und RETURN

zu laden.

Nach der Titelgrafik springt das Programm in das Inhaltsverzeichnis des Magazins. Hier können Sie mit der Leertaste weiter- und mit SHIFT und Leertaste zurückblättern. Mit RETURN wird das angezeigte Programm ausgewählt und geladen.

Unser Betriebssystem stellt neben dem Inhaltsverzeichnis noch weitere Funktionen zur Verfügung. Diese werden mit der CTRL-Taste und einem Buchstaben aufgerufen. Sie brauchen sich eigentlich nur CTRL und H zu merken, denn mit dieser Tastenkombination erscheint eine Hilfsseite auf dem Bildschirm, die alle weiteren System-Befehle enthält. Nicht immer sind alle Optionen möglich. Befehle, die zur Zeit gesperrt sind, werden auf der Hilfsseite dunkel angezeigt. Hier nun die Befehle im einzelnen:

CTRL und H

Haben wir schon erwähnt – damit wird die Hilfsseite ein- und ausgeschaltet.

CTRL und I

Sie verlassen das gerade laufende Programm und kehren ins Inhaltsverzeichnis zurück.

CTRL und F

Ändert die Farbe des Bildschirmhintergrundes. Diese Option funktioniert immer, wenn ein Programm läuft oder Sie sich im Inhaltsverzeichnis befinden, aber nicht auf der Hilfsseite.

CTRL und R

Wie CTRL-F, wirkt auf die Rahmenfarbe.

CTRL und B

Sie erhalten einen Ausdruck der Textseite eines laufenden Programmes auf einem angeschlossenen Drucker. Diese Hardcopy-Routine ist angepaßt für Commodore-Drucker und kompatible Geräte. Das Programm wählt automatisch die richtige Geräteadresse (4, 5 oder 6) aus. Sie können diese Routine mit der ←-Taste abbrechen.

CTRL und S

Programme, die auch außerhalb von INPUT 64 laufen, können Sie mit diesem Befehl auf eine eigene Diskette überspielen. Das sind in diesem Special alle Programme der zweiten Disketten-Seite. Wenn Sie diesen Befehl aktivieren, bekommen Sie unten auf der Hilfsseite angezeigt, wie viele Blocks das File auf der Diskette belegen wird. Geben Sie nun den Namen ein, unter dem das Programm auf Ihre Diskette geschrieben werden soll. In der Regel handelt es sich um Programme, die Sie ganz normal laden und mit RUN starten können. Ausnahmen sind in den jeweiligen Programmbeschreibungen erläutert.

CTRL und D

Gibt das Directory der eingelegten Diskette aus. Die Ausgabe kann mit der Leertaste angehalten und mit RETURN wieder fortgesetzt werden. Ein Abbruch ist mit der ←-Taste möglich. Wenn das Directory vollständig ausgegeben ist, gelangen Sie mit der RETURN-Taste zurück ins unterbrochene Programm beziehungsweise auf die Hilfsseite.

CTRL und @

Disk-Befehle senden, zum Beispiel Formatieren einer neuen Diskette oder Umbenennen eines Files. Für den zu sendenden Befehls-String gilt die übliche Syntax, natürlich ohne ein- und ausführende Hochkommata. CTRL-@ und RETURN gibt den Zustand des Fehlerkanals der Floppy auf dem Bildschirm aus. Weiter im Programm oder zurück auf die Hilfsseite führt ein beliebiger Tastendruck.

CTRL und A

Sucht auf der Diskette nach einem INPUT 64-Inhaltsverzeichnis. Mit diesem Befehl ist es möglich, ohne den Rechner auszuschalten, auf die andere Disketten-Seite „umzusteigen" (oder Programme von anderen INPUT 64-Disketten zu laden!). Das funktioniert aber nur bei den Ausgaben ab 4/86.

Bei Ladeproblemen

Bei nicht normgerecht justiertem Schreib-/Lesekopf oder bei bestimmten Serien wenig verbreiteter Laufwerke (1570) kann es vorkommen, daß das ins INPUT-Betriebssystem eingebaute Schnelladeverfahren nicht funktioniert. Eine mögliche Fehlerursache ist ein zu geringer Abstand zwischen Floppy und Monitor/Fernseher. Das Magazin läßt sich auch im Normalverfahren laden, eventuell lohnt sich der Versuch:

LOAD "LADER",8,1

Sollte auch dies nicht zum Erfolg führen, senden Sie bitte die Diskette mit einem kurzen Vermerk über die Art des Fehlers und die verwendete Gerätekonstellation an den Verlag (Adresse siehe Impressum).

SPECIAL 2

Liebe 64er-BesitzerInnen!

Ob bei einer Single die A-Seite oder die B-Seite der wirkliche Hit ist, stellt sich oft erst im nachhinein heraus. Letzten Endes ist das eine Sache des persönlichen Geschmacks.

So ähnlich ist es auch mit dem Ihnen hier vorliegenden „Special 2" des elektronischen Magazins INPUT 64. Auf zwei Diskettenseiten sind alle wichtigen Programme aus der „Assembler-Ecke", die in INPUT 64 veröffentlicht wurden, zusammengefaßt. Auf der A-Seite befindet sich ein in dieser Art einzigartiges Software-Projekt: ein dialogorientiertes, interaktives Lernprogramm zum Thema „6502-Maschinensprache". Mit dieser aus sechs Kurseinheiten zusammengesetzten Assembler-Schule wird die Software dazu eingesetzt,

- Ihre Lernerfolge zu überwachen,
- Ihnen Hilfestellungen anzubieten
- und Ihnen die Möglichkeit zu geben, direkt am Rechner Ihre ersten Programme kontrolliert auszuprobieren.

Dieser Lehrgang behandelt alles, was man zum Programmieren in Maschinensprache wissen muß. Er ist in wichtigen Teilen so geschrieben, daß grundlegende Erkenntnisse auch auf andere Rechner übertragbar sind – nach dem Motto: Verstehste einen, verstehste alle!

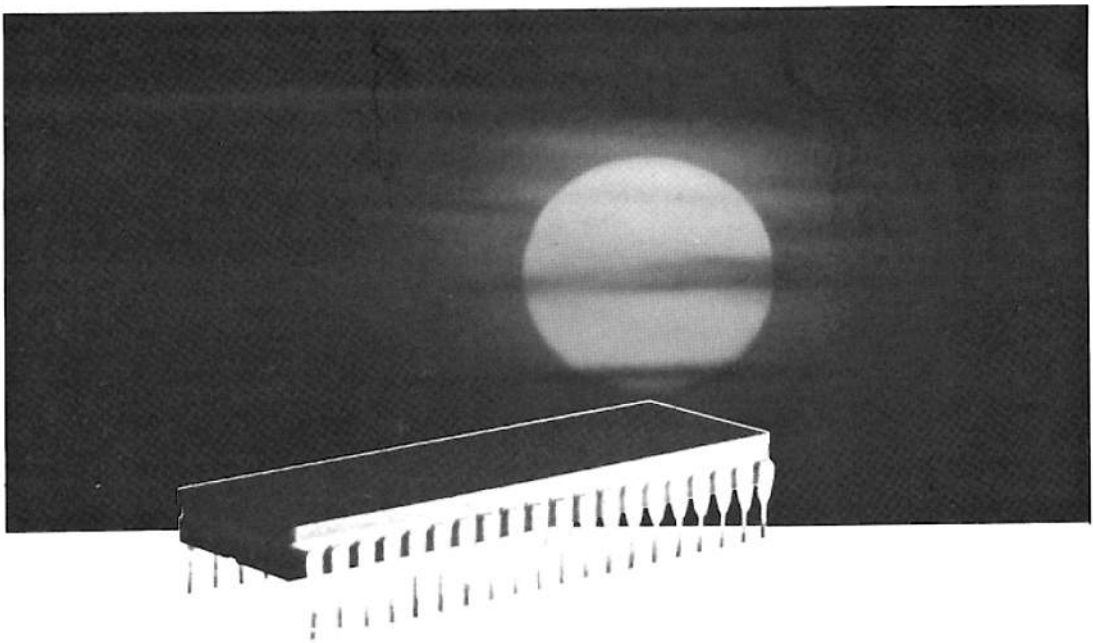
Die Software auf der B-Seite der Diskette stellt alle Werkzeuge zur Verfügung, die zur komfortablen Programmentwicklung notwendig sind. All diese Programme verdienen unseres Erachtens das Etikett „professionelle Software" – was beileibe nicht heißen soll, daß nur Profis damit umgehen können. Im Gegenteil: gerade die ausgereifte Konzeption und die praxisgerecht aufeinander abgestimmten Funktionen machen es auch dem Anfänger möglich, unbeschwert von Unzulänglichkeiten der Entwicklungswerkzeuge seine ersten „Gehversuche" zu starten.

Falls trotzdem noch die eine oder andere Frage durch die Dokumentation unbeantwortet bleibt – nobody is perfect –, können Sie wie gewohnt jeden Dienstag von 9 bis 16.30 Uhr unsere telefonische Leserberatung in Anspruch nehmen (Tel.: 05 11/ 53 52-141).

Jürgen Seeger

J. Seeger

INHALT



Dem Prozessor auf der Spur

Die INPUT 64-Assembler-Schule

Mit unserem neuen Kurs über 6502-Maschinensprache wollen wir Sie aus dem Dunst dieser Gerüchteküche befreien. Steigen Sie ein in die 'Haute Cuisine' der Assembler-Programmierung. Im Verlauf dieses Kurses werden Sie köstliche Rezepte kennenlernen, mit denen Sie Ihren Rechner füttern können. Eine Versuchsküche, in der garantiert nichts anbrennt, liefern wir Ihnen gratis dazu.

Wenn man normalerweise anfängt, sich mit der Maschinensprache zu beschäftigen, ist spätestens nach zwei Wochen ein neuer Reset-Knopf oder gar ein neuer Netzschalter für den Rechner fällig. Ganz abgesehen von den Nerven, die durch ständiges neues Laden von Assembler und Monitor auf eine harte Probe gestellt werden.

Mit der INPUT-Assembler-Schule geht das ganz anders: In dem Programm enthalten ist in jeder Folge ein Editor, mit dem Sie eigene Maschinenprogramme erstellen können. Schon in dieser Phase werden Syntax-Fehler abgefangen. Das selbsterstellte Programm kann dann von dem ebenfalls mitgelieferten Assembler übersetzt werden.

Maschinensprache ist unwahrscheinlich kompliziert, schwer zu lernen, Maschinensprache-Programme sind aufwendig zu erstellen, und überhaupt ist Maschinensprache nur etwas für Profis. Diese und andere Vorurteile halten sich immer noch hartnäckig in den Köpfen vieler Computer-Besitzer. Und trotzdem träumt fast jeder davon, diese sagenumwobene Sprache zu lernen, denn überall ist zu hören, daß Maschinensprache die schnellste Programmiersprache überhaupt ist und erst sie die letzten Möglichkeiten des Computers aus ihm herausholt.

Der eigentliche Clou ist aber der Prozessor-Simulator. Er ermöglicht das Austesten der eigenen Programme unter vollständiger Kontrolle des Rechners. Der Einzelschritt-Modus knackt selbst die hartnäckigste Endlos-Schleife, und kein undokumentierter Opcode kann den Rechner zum Absturz bringen.

Die Bedienung des Simulators ist auf Seite 5 unter der Überschrift „Ausprobieren und Simulieren" erklärt.

Die Flachsprache

Maschinensprache ist die 'Muttersprache' eines jeden Computers. Darum ist sie so schnell. Alle höheren Programmiersprachen wie etwa BASIC brauchen einen Übersetzer, der die Programme für den Prozessor verständlich macht. Auch im C64 läuft ein solcher Übersetzer. Es handelt sich dabei um ein Maschinensprache-Programm (was sonst?), genannt BASIC-Interpreter. Er liest ein BASIC-Programm Befehl für Befehl, entschlüsselt jeden einzelnen und führt die entsprechenden Aktionen aus.

Bei Maschinensprache-Programmen ist das ganz anders: Hier versteht der Prozessor direkt die im Speicher liegenden Befehle. Es ist wohl klar, daß so ein 'fest verdrahteter' Befehls-Ausführer schneller ist als ein Programm, das die Befehle erst in Maschinensprache übersetzen muß.

Andererseits sind die Befehle, die der Prozessor direkt versteht, nicht so leistungsfähig wie beispielsweise BASIC-Befehle. Für ein Maschinensprache-Programm muß man also das zu lösende Problem weiter 'aufdröseln'.

Vokabeln

Wir haben gesagt, daß die Maschinensprache-Befehle so im Speicher liegen, daß der Prozessor sie direkt verstehen kann. Für uns Menschen sind sie aber nur schwer zu merken. Darum hat man die sogenannte Assembler-Sprache entwickelt. Sie besteht aus Befehlen, die sehr eng mit der Maschinensprache in Zusammenhang stehen. Jedem Maschinensprache-Befehl ist eine leicht zu merkende Kombination aus drei Buchstaben – genannt Mnemonic zugeordnet (jawohl, mit 'Mn' – ich habe das Wort nicht erfunden). Wir werden uns in diesem Kurs daher nicht mit der eigentlichen Maschinensprache auseinandersetzen, sondern mit der Assemblersprache.

Es gibt Programme, die ein in Assemblersprache geschriebenes Programm in ein Maschinen-Programm übersetzen. So ein Programm heißt Assembler. Den Text, den es verarbeitet, nennt man Source- oder Quell-Code, das Maschinen-Programm, das er erzeugt, heißt dann Object-Code. Allerdings sollte man diese strenge Begriffstrennung nicht allzu eng sehen. Man spricht oft schon von einem Assembler- oder Maschinen-Programm, wenn man eigentlich den Quell-Code meint. Oft wird auch leichthin behauptet, ein Maschinensprache-Programm sei 'in Assembler geschrieben'.

Von Bits und Bytes

Ein Bit ist die kleinste Informationseinheit eines jeden Digital-Rechners. Man kann sich ein Bit als einen Schalter vorstellen, der entweder offen ist (dann fließt kein Strom, das Bit hat den Wert Null) oder geschlossen (Stromfluß, Wert ist Eins). In solchen Schaltern kodiert jeder Computer seine Programme und Daten. Einen Code mit nur zwei Zuständen nennt man binären oder zweiwertigen Code. Der Name Bit kommt übrigens aus dem Englischen und ist die Abkürzung für BInary digiT (binäre Ziffer).

Aus mehreren binären Ziffern kann man eine Zahl bilden. Eine vierstellige Binärzahl (auch Nibble genannt) kann sechzehn verschiedene Werte annehmen. Das zeigt Tabelle 1. Auf die dritte Spalte kommen wir noch zu sprechen.

Man kann mit einem Nibble also die Zahlen Null bis Fünfzehn darstellen. Dabei hat jede Stelle einer Binärzahl den doppelten Stellenwert der rechts von ihr stehenden. In dem Ihnen wahrscheinlich geläufigeren Dezimalsystem, bei dem jede Ziffer zehn verschiedene Werte annehmen kann, hat jede Stelle ja auch den zehnfachen Wert der vorangehenden.

Der Prozessor, der im C64 'tickt', ist ein Acht-Bit-Prozessor. Das heißt, daß er Register (interne Speicherzellen) besitzt, die acht Bits parallel verarbeiten können. Eine solche Binärzahl mit acht Stellen heißt Byte.

Um nicht jedesmal eine achtstellige Binärzahl hinschreiben zu müssen, wenn man ein Byte meint, haben findige Mathematiker das Hexadezimalsystem erfunden. Bei ihm gibt es sechzehn verschiedene Ziffern. Ein Nibble läßt sich also mit nur einer, ein Byte mit zwei Hexadezimalziffern darstellen. Neben den gewohnten Ziffern werden noch die Buchstaben A bis F als Ziffern 'mißbraucht'. Die Zuordnung zeigt Tabelle 1.

Im Hexadezimalsystem hat jede Ziffer den sechzehnfachen Wert der vor ihr stehenden. Mit zwei Nibbles kann man also 16*16, das sind 256 verschiedene Zahlen, darstellen (0 bis 255).

Harte Sachen

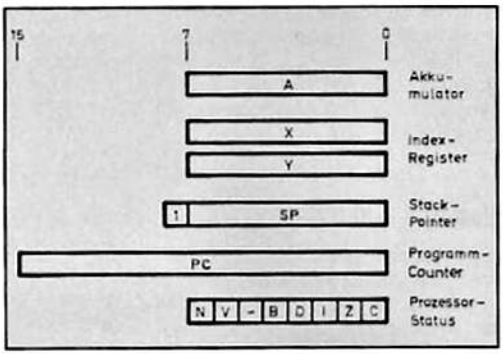
Bevor es mit dem Programmieren richtig losgeht, sollen noch einige Begriffe erwähnt werden: Zum Beispiel, daß der Prozessor des C64 den Namen 6510 trägt. Er ist ein Abkömmling der 6502-CPU und versteht die gleichen Befehle wie diese. CPU ist mal wieder eine Abkürzung aus dem Englischen und kommt von 'Central Processing Unit', etwa 'Zentrale Verarbeitungseinheit'. Damit ist auch die Aufgabe eines Prozessors innerhalb eines Computersystems umrissen.

Binär	Dezimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Tabelle 1:
Die Zahlensysteme auf einen Blick

Zu einem Rechner gehören noch die beiden Funktionsblöcke Speicher und Ein-/Ausgabe-Einheit. Mit ihnen steht der Prozessor über elektrische Leitungen in Verbindung, die man Busse nennt. Es gibt einen acht Bit 'breiten' Datenbus, über den die Informationen zwischen den Komponenten ausge-

Bis auf den Programmzähler sind die Register des 6510 acht Bit breit. In der ersten Folge der Assembler-Schule beschäftigen wir uns mit dem Akkumulator und einem Flag im Statusregister.



tauscht werden, und einen Adreß-Bus, mit dem die CPU dem Speicher mitteilt, welche Speicherstelle gemeint ist. Darüber hinaus gibt es noch Leitungen, die den zeitlichen Ablauf der Aktionen oder die Richtung des Datenflusses bestimmen. Diese Leitungen faßt man zum Kontroll- oder Steuer-Bus zusammen.

Der Adreß-Bus umfaßt beim 6510 sechzehn Leitungen. Damit können zwei hoch sechzehn, das sind 65536 verschiedene Speicherzellen angesprochen werden. Diesen Bereich teilt man in 256 Seiten (Pages) zu je 256 Adressen ein. Wir werden noch sehen, daß einige dieser Seiten für den Prozessor eine besondere Bedeutung haben.

Das erste Programm

Jetzt wollen wir aber endlich programmieren! Dabei soll unser Prozessor gleich etwas tun, was man täglich benötigt. Wir wollen ein Programm schreiben, das zwei Zahlen addiert. Der erste Assembler-Befehl, den wir benutzen wollen, heißt

LDA #6

Wie fast alles, was mit Computern zu tun hat und nicht auf Anhieb verständlich ist, handelt es sich um eine englische Abkürzung. Sie steht für 'LoaD Accumulator'. Dieser Befehl dient dazu, eine jener CPU-internen Speicherzellen, die wir Register nennen, mit einem Wert zu füllen. Dieses Register heißt Accumulator (Sammler) oder kurz Akku. Es ist ein sehr wichtiges Register, da es bei allen arithmetischen Operationen beteiligt ist.

Der Prozessor will natürlich noch wissen, welchen Wert er in den Akku laden soll. In unserem Beispiel soll das die Zahl 6 sein. Das Doppelkreuz signalisiert, daß es unmittelbar (immediate) die Zahl 6 sein soll und nicht etwa der Inhalt der Speicherstelle Nummer 6.

Als nächstes soll der Prozessor zu der im Akku gespeicherten Zahl etwas addieren, beispielsweise die Zahl 3. Der Befehl dazu lautet

ADC #3

und bedeutet 'ADd to accumulator with Carry'. Das Doppelkreuz und die 3 stellen wieder ein unmittelbares Argument dar.

Was soll nun 'with Carry' bedeuten? Dazu muß man wissen, daß es in der CPU ein sogenanntes Statusregister gibt. Eines seiner Bits heißt Carry-Flag. Diese 'Flagge' wird von der CPU unter anderem dann gesetzt, wenn bei einer Addition ein Übertrag entsteht. Übertrag heißt, daß das Ergebnis größer als 255 wurde und somit nicht mehr in ein 8-Bit-Register paßt. Sein Inhalt wird auch bei einer Addition immer noch zum Ergebnis dazugezählt. Wir hätten es darum vor der Addition löschen sollen. Dazu dient der Befehl

CLC

(CLear Carry flag), den wir also noch vor unser Programm schreiben müssen. Bislang besteht unser Programm also aus den Befehlen

```
CLC
LDA #6
ADC #3
```

Laß' gehn!

Wir wollen uns jetzt auf dem Bildschirm ansehen, wie es arbeitet. Gehen Sie dazu ins Hauptmenü der 'Assembler-Schule', drücken Sie F3 und wählen Programm 1 aus. Sie sehen (hoffentlich) ein Listing unseres ersten Programms. Drücken Sie nun F7. Am oberen Bildschirmrand erscheinen die Zeilen

```
A  X  Y  SP  PC    NV-BDIZC
00 00 00 fe  c000  00110000  clc
```

Ganz links, unter dem A, wird der Inhalt des Akkus (hexadezimal) angezeigt, unter den Buchstaben NV-BDIZC in binärer Form der des Status-Registers. Sein niederwertigstes Bit – unter dem C – ist die Carry-Flagge. Rechts steht der als nächstes auszuführende Befehl. Um ihn ausführen zu lassen, drücken Sie die Leertaste.

An den Register-Inhalten ändert sich dadurch nichts. Die Carry-Flagge war ja schon gelöscht, aber sicher ist sicher.

Nach dem nächsten Druck auf die Leertaste wird der Befehl LDA #6 ausgeführt. Der Akku enthält nun die Zahl 6. Noch mal SPACE – die Addition wird ausgeführt, der Akku enthält nun 9. Die Carry-Flagge zeigt nach wie vor 0 an, ein Übertrag ist ja nicht aufgetreten.

In den Speicher schreiben

Der nächste Befehl, den Sie lernen sollten, heißt

STA $C008

und bedeutet 'STore Accu', also 'speichere den Inhalt des Akkus ab'. Und zwar in die Speicherstelle, deren Adresse hinter dem Befehl steht. In unserem Beispiel also an die Adresse C008 hexadezimal.

Springen Sie mit F7 wieder in den Editor, schreiben Sie diesen Befehl hinter unser bisheriges Programm und lassen es erneut ausführen. In der zweiten inversen Zeile in der unteren Hälfte des Bildschirms sollten Sie hinter der Adresse $C008 beobachten können, wie der Inhalt des Akkus dorthin kopiert wird.

Sie könnten erneut in den Editor gehen und andere Zahlen hinter die Befehle LDA und ADC schreiben. Die Doppelkreuze sollten Sie aber stehenlassen. Probieren Sie doch mal aus, wie groß die beiden Argumente höchstens sein können, ohne daß nach der Addition die Carry-Flagge gesetzt ist. Wie sieht das Ergebnis aus, wenn ein Übertrag auftritt?

Noch'n Programm

Wenn Sie diese Fragen geklärt haben, laden Sie bitte das zweite Programm in den Editor. (Mit STOP ins Hauptmenü, F3 drücken und 2 wählen.) Dieses Programm soll Ihnen etwas über Programm-Dokumentation zeigen.

Alle Zeilen, die mit einem Semikolon beginnen, enthalten einen Kommentar, der einzig und allein der Lesbarkeit des Programms dienen soll. Sie treten nach der Übersetzung durch den Assembler im eigentlichen Maschinen-Programm nicht mehr in Erscheinung.

Überall, wo in der zweiten Spalte (mit der Überschrift 'labl') etwas steht, wird ein Label definiert. Labels sind symbolische Namen für bestimmte Speicheradressen. Man kann sie auch als Variable für den Assembler auffassen. Sie enthalten als Wert die Adresse, hinter der sie stehen. Überall, wo im Quell-Code beispielsweise das Argument 'SUM1' auftaucht, setzt der Assembler dafür $C014 ein. Der Sinn dieser symbolischen Adressen ist in dem Artikel „Assembler als Hochsprache" an anderer Stelle in diesem Heft ausführlich erklärt.

Doch nun zu dem, was das Programm tut: Den Befehl CLC kennen Sie schon. Der nächste Befehl heißt

LDA SUM1

SUM1 steht für 'erster Summand'. Dieser Name ist aber völlig beliebig. Er ist wie gesagt nur ein symbolischer Name für die Adresse $C014. Vor diesem Argument steht kein Doppelkreuz. Es handelt sich also nicht um ein unmittelbares Argument. (Die Zahl $C014 würde im übrigen auch gar nicht in den Akku passen.) Diese Adressierungsart heißt 'absolut'. Das bedeutet, der Akku soll mit dem Inhalt der absoluten Adresse $C014 geladen werden.

An der Adresse $C014 steht

W $1234

W ist kein Maschinensprache-Befehl, sondern eine Anweisung für den Assembler. Sie veranlaßt ihn, bei der Übersetzung an diese Adresse die dahinterstehende Zwei-Byte-Zahl zu schreiben. Dabei schreibt er das niederwertige Byte an die angegebene und das höherwertige an die folgende Adresse. Eine Zwei-Byte-Zahl in dieser Form wird auch 'Wort' genannt – daher der Befehl W.

In unserem Beispiel werden also nach der Übersetzung die Speicherstelle $C014 den Inhalt $34 und die Speicherstelle $C015 den Wert $12 haben.

Nach der Ausführung des Befehls 'LDA SUM1' wird der Akku also in unserem Beispiel den Inhalt $34 haben.

Die folgenden Befehle sollten Ihnen keine Rätsel mehr aufgeben: ADC SUM2 addiert den Inhalt der Speicherstelle, die den symbolischen Namen 'SUM2' hat, zum Inhalt des Akkus. STA SUMM speichert den Akku-Inhalt an die Adresse mit dem Label 'SUMM'.

An dieser Adresse taucht eine andere Assembler-Anweisung auf: Der Befehl

S 2

bewirkt, daß der Assembler an dieser Adresse zwei Bytes Platz (Space) läßt. Er füllt diese Adressen mit Null. Diese Adressen können vom Programm als Variable benutzt werden. In unserem Fall nehmen Sie das Ergebnis einer Zwei-Byte-Addition auf.

Der letzte unklare Befehl in diesem Programm ist

BRK

Ausprobieren und Simulieren

Zur Bedienung des Lernprogramms

Die INPUT 64-Assembler-Schule setzt sich aus mehreren Teilen zusammen. Nach dem Laden sehen Sie ein Titelbild, von dem aus Sie mit einem beliebigen Tastendruck in das Hauptmenü gelangen.

Wenn Sie nun F1 drücken, gelangen Sie in ein Menü, das Ihnen verschiedene Themen zur Auswahl stellt. Die Erklärungen, die Sie jetzt abrufen können, sollten Sie parallel zum Beiheft lesen. Beide Medien ergänzen sich hier. Sie können die Erklärungen auch mit CTRL-b ausdrucken. Ins Hauptmenü gelangen Sie jederzeit mit der STOP-Taste zurück.

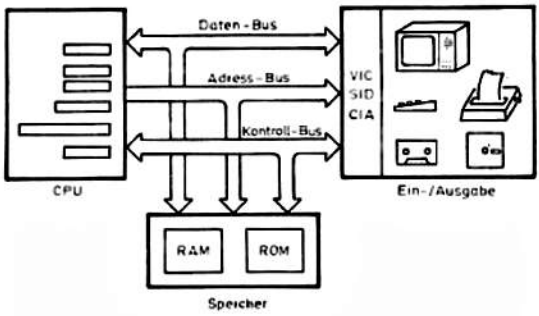
Mit F3 gelangen Sie aus dem Hauptmenü zu einer Auswahl verschiedener Beispielprogramme. Sie können eines davon auswählen, das Sie sich dann im Editor anschauen oder auch verändern können. Wenn Sie an dieser Stelle eine Null eingeben, enthält der Editor das zuletzt bearbeitete Programm, beim ersten Aufruf ist der Textspeicher leer.

Wenn Sie ein Beispielprogramm bearbeitet haben und – mit der STOP-Taste – wieder ins Hauptmenü springen, können Sie Ihren Text auch auf einen Drucker ausgeben lassen oder auf einen eigenen Datenträger abspeichern. Abgespeicherte Programme können Sie direkt mit dem INPUT-ASS (Ausgabe 6/86) laden und weiterbearbeiten.

Vom Editor aus gelangen Sie mit F7 in einen integrierten Simulator. Hier können Sie unsere Programmbeispiele oder Ihre selbstentworfenen Programme ablaufen lassen und testen, ob sie sich erwartungsgemäß verhalten.

Ausführliche Hinweise zur Bedienung des Editors und des Simulators sind im Programm enthalten. Sie können sie von dort aus jeweils mit der Funktionstaste F6 aufrufen. Es wird empfohlen, diese Seiten vor der Benutzung des Programmpakets einmal gründlich zu lesen. Besitzer eines Druckers können sie auch mit CTRL-b zu Papier bringen.

Jedes Computersystem besteht aus drei Komponenten: einer zentralen Verarbeitungseinheit (CPU), einem Speicher und einer Ein-/Ausgabe-Einheit. Den Kontakt mit der Außenwelt stellen beim C64 ein Video-Controller, ein Synthesizer-Baustein und zwei universelle Ein-/Ausgabe-Chips her.



Er bedeutet BReaK (Abbruch). Wenn der Prozessor auf diesen Befehl trifft, hört er mit der Bearbeitung des Programmes auf und springt an eine (hardwaremäßig festgelegte) Adresse. Beim C64 wird dann normalerweise der Bildschirm gelöscht, und in der ersten Bildschirmzeile erscheint die Meldung 'READY.' Innerhalb unseres Simulators wird diese Meldung in der untersten Bildschirmzeile ausgegeben, und Sie können wieder in den Editor oder zum Hauptmenü springen.

Zieh ab!

Im dritten Beispiel-Programm geht es um die Subtraktion. Hier erst mal die neuen Befehle:

SEC

ist die Abkürzung für 'SEt Carry flag' und tut genau das Gegenteil von CLC. Mit diesem Befehl kann – zum Beispiel vor einer Subtraktion – in die Carry-Flagge eine 1 geschrieben werden.

SBC #$82

subtrahiert das Argument vom Inhalt des Akkus. Ausgeschrieben heißt der Befehl 'SuBtract from accu with Carry'. Dabei muß die Carry-Flagge gesetzt sein, sonst wird 'einer mehr' abgezogen. Tritt ein Unterlauf auf, so wird die Carry dabei gelöscht.

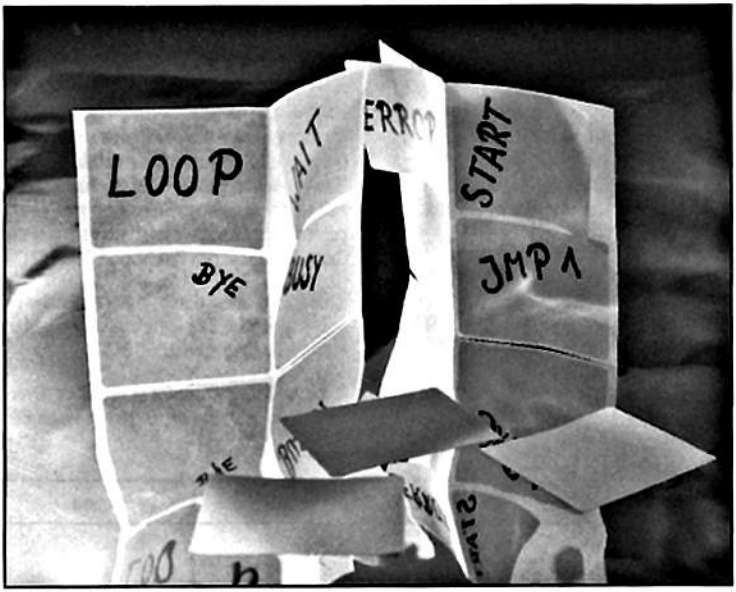
Im Prinzip funktioniert eine Subtraktion in Assembler genau wie eine Addition. Man muß nur die Befehle ADC gegen SBC und CLC gegen SEC austauschen.

Das Beispiel-Programm Nummer drei zeigt, wie ein Ein-Byte-Wert von einer Zwei-Byte-Zahl abgezogen wird. Dabei treten in einem Programm verschiedene Adressierungsarten auf – das ist völlig normal, die beiden anderen Programme waren etwas Besonderes.

Der Befehl SBC #0 sieht auf den ersten Blick einigermaßen überflüssig aus, ist er aber nicht. Wenn nämlich in der vorhergehenden Subtraktion vom Low-Byte ein Unterlauf aufgetreten ist, ist die Carry-Flagge an dieser Stelle gelöscht, und es wird nicht Null, sondern Eins vom High-Byte abgezogen. Und genau das ist ja beabsichtigt. Mit diesem 'Trick' sind Ein- und Zwei-Byte-Werte verknüpfbar. Diese Vorgehensweise entspricht dem spaltenweisen Subtrahieren von Dezimal-Zahlen, in der ein Unterlauf ja auch nach links übertragen wird.

Auch bei diesem Programm kann man übrigens die jeweils entgegengesetzten Befehle einsetzen, und es wird addiert. Wenn nämlich bei der Addition auf das niederwertige Byte ein Überlauf entsteht, wird die Carry gesetzt, und der Befehl ADC #0 addiert in Wirklichkeit Eins.

In der nächsten Folge der 'Assembler-Schule' werden Sie unter anderem lernen, wie negative Zahlen in Assembler behandelt werden. HS



Die entscheidenden Sprünge

Zweite Folge der Assembler-Schule

Bevor wir aber zu neuen Programmen kommen, wollen wir einen einen kurzen Blick auf die Art und Weise werfen, wie Maschinen-Programme im Speicher abgelegt und vom Prozessor verarbeitet werden.

Ihnen ist sicherlich beim Experimentieren schon aufgefallen, daß die Adressen, an denen die einzelnen Befehle liegen, ziemlich unregelmäßig durchnumeriert werden. Das liegt daran, daß die Befehle unterschiedlich viel Platz im Speicher benötigen. Die 6502-CPU kennt Befehle, die ein, zwei oder drei Bytes belegen. Diese Instruktionslänge ist abhängig von der Adressierungsart. So sind die Befehle mit implizierter Adressierung (Das sind die ohne Argument.) nur ein Byte lang, bei unmittelbarer Adressierung (gekennzeichnet durch ein Doppelkreuz) werden zwei und bei absoluter drei Bytes belegt.

Nachdem in der ersten Folge einige grundlegende Maschinensprache-Befehle gezeigt wurden, befassen wir uns in dieser Folge mit weiteren Arithmetik-Operationen. Neben neuen Befehlen werden Sie die Bits des Statusregisters im einzelnen kennenlernen sowie die Index-Register X und Y benutzen. Wenn Sie diese Folge durchgearbeitet haben, werden Sie außerdem wissen, wie bedingte Programm-Sprünge in Assembler kodiert werden.

Programme sind auch nur Daten

Der eigentliche Befehl steckt dabei immer im ersten Byte. In ihm ist auch die Instruktionslänge verschlüsselt. Die beiden Versionen des LDA-Befehles, die Sie bislang kennen, haben zum Beispiel auch unterschiedliche Instruktions-Bytes. Bei der Immediate-Adressierung ist es $A9 und bei absoluter Adressierung $AD. Dadurch erkennt der Prozessor die unterschiedlicheAdressierungsart und damit auch die Instruktionslänge.

Wenn der Prozessor beginnt, ein Programm abzuarbeiten, liest er das erste Byte. Bei Bedarf werden dann noch ein oder zwei Bytes gelesen, der Befehl ausgeführt, und das Spiel wird mit dem ersten Byte des nächsten Befehls fortgeführt.

Normalerweise wird also ein Byte nach dem anderen gelesen und die entsprechende Aktion durchgeführt. Sie werden in dieser Folge aber noch Kommandos kennenlernen, mit denen man diese Reihenfolge beeinflussen kann. Eines kennen Sie schon, nämlich BRK. Trifft der Prozessor auf diesen Befehl, macht er an einer speziellen Adresse weiter. Diese Adresse ist innerhalb des ROMs. Das Programm, das dort steht, löscht normalerweise den Bildschirm, gibt das Wort READY aus und wartet auf eine Eingabe.

Aus dem bisher gesagten erkennt man (hoffentlich), daß ein Maschinenprogramm im Prinzip nichts anderes ist als eine Folge von Bytes, die hintereinander im Speicher liegen. Mit dem Ende eines Programmes wird aber nicht das Ende des Speichers erreicht. Wenn ein Programm also einfach endet, ohne mit einem BRK oder einem anderen geeigneten Befehl (Sie werden noch verschiedene kennenlernen) abgeschlossen zu werden, versucht der Prozessor, die folgenden Bytes auch als Programm zu interpretieren und auszuführen. Dabei entstehen die verrücktesten Effekte, meistens endet so etwas mit einem 'Absturz' des Rechners.

Innerhalb unseres Simulators kann nichts passieren, Sie erhalten eine entsprechende Fehlermeldung. Wenn Sie aber ein Programm 'im richtigen Leben' laufen lassen, das in Simulator mit der Meldung „Programmende" oder „Opcode nicht ausführbar" abbricht, sollten Sie sich über nichts wundern. Kaputtgehen kann der C64 dabei nicht, aber Sie werden wahrscheinlich einen Reset-Taster oder gar den Netzschalter betätigen müssen.

Genauso, wie man Bytes im Speicher entweder als Programm oder als Daten auffassen kann, gibt es für Datenbytes meistens verschiedene Interpretationsmöglichkeiten.

Weniger als nichts

In der ersten Folge haben wir unter anderem gelernt, wie Ein- und Zwei-Byte-Werte in Maschinensprache subtrahiert werden. Dabei diente die Carry-Flagge des Prozessor-Status-Registers als Anzeige für einen Übertrag in die nächsthöhere Stelle beziehungsweise für einen Fehler. Wenn bei der Subtraktion eine negative Zahl herauskam, sind wir davon ausgegangen, daß ein Fehler vorliegt. Mit Hilfe des sogenannten Zweierkomplementes ist es nun möglich, auch negative Ergebnisse sinnvoll zu interpretieren. Dabei werden alle Bytes mit gesetztem höchstwertigem Bit als negative Zahlen aufgefaßt. Das genaue Umrechnungsverfahren ist im Programm beschrieben.

Je nachdem, wie die Ergebnisse einer Rechnung weiterverarbeitet werden, kann man also die Zahlen $80 bis $FF einfach umrechnen und ihnen die Werte 128 bis 255 zuordnen oder mit Hilfe des Zweierkomplementes als die negativen Zahlen -128 bis -1 auffassen. Diese Tatsache wird durch den Zahlenkreis (siehe Seite 9) anschaulich verdeutlicht.

Bei letzterer Darstellungsweise kann es natürlich auch zu Über- beziehungsweise Unterschreitungen des gültigen Zahlenbereiches kommen. Addiert man beispielsweise die Zahlen 73 und 58, so ist das Ergebnis größer als 127, das höchstwertige Bit ist gesetzt, und daher wäre das Resultat als negative Zahl zu interpretieren. Zum Glück stellt der Prozessor auch für diesen Fall eine Möglichkeit der Bereichsüberprüfung zur Verfügung. In dem obigen Beispiel wäre nach der Ausführung der Addition die Overflow-Flagge im Statusregister gesetzt. Dieses Bit – auch V-Flag genannt – wird automatisch bei jeder Addition und Subtraktion mitversorgt, ähnlich wie die Carry-Flagge. Sein Inhalt hat aber auf eventuell folgende ADC- oder SBC-Befehle keinen Einfluß.

Hier die Regeln, nach denen der Prozessor die V-Flagge setzt: Die CPU geht immer davon aus, daß sie es mit vorzeichenbehafteten Zahlen zu tun hat. Die V-Flagge wird gesetzt, wenn zu einer positiven Zahl eine positive Zahl addiert oder von ihr eine negative Zahl subtrahiert wird und dabei das Ergebnis größer als 127, also negativ, ist. Im negativen Bereich ist es ebenso: Negative Zahl plus negative Zahl oder negative Zahl minus positive Zahl und positives Ergebnis ergibt gesetztes Overflow-Flag.

Das hört sich zwar sehr kompliziert an, ist es aber eigentlich gar nicht, denn es geschieht völlig automatisch. Wenn man mit vorzeichenbehafteten Zahlen rechnet, sollte man nach einem ADC- oder SBC-Befehl die V-Flagge prüfen. Sie können sich einfach darauf verlassen, daß das Ergebnis stimmt, wenn sie nicht gesetzt ist. Wie man ein Flag des Status-Registers innerhalb eines Programmes prüft, dazu kommen wir gleich.

Statusfragen

Zunächst wollen wir einen ausführlichen Blick auf die einzelnen Bits des Prozessor-Status-Registers – auch kurz P-Register genannt – werfen (siehe Seite 10). Es ist wie die meisten 6502-Register acht Bits breit. Jedoch kann man es nicht wie zum Beispiel den Akku direkt manipulieren. Vielmehr werden einzelne Bits bei der Programm-Ausführung automatisch beeinflußt. Wie wir schon gesehen haben, können einzelne Bits auch vom Programmierer direkt gesetzt oder zurückgesetzt werden wie die Carry-Flagge mit den Befehlen SEC und CLC.

Jedes Flag wird durch einen Buchstaben gekennzeichnet. Die Bezeichnungen lauten von links nach rechts NV-BDIZC

Die Bits 0 (C = Carry-Flag) und 6 (V, Overflow) hatten wir schon. Bit 5 hat keine Bedeutung und ist Hardware-bedingt immer gesetzt.

Mit den bisher bekannten Befehlen haben wir die Bits 1 und 7 auch schon beeinflußt, aufmerksame Beobachter haben es vielleicht beim Experimentieren bemerkt.

Bit 1 heißt Zero- oder kurz Z-Flagge. Der Prozessor setzt sie immer, wenn das Er-

Opcode	Bedeutung	Bedingung
BPL	Branch on result PLus	N-Flag = 0
BMI	Branch on result MInus	N-Flag = 1
BVC	Branch on oVerflow Clear	V-Flag = 0
BVS	Branch on oVerflow Set	V-Flag = 1
BCC	Branch on Carry Clear	C-Flag = 0
BCS	Branch on Carry Set	C-Flag = 1
BNE	Branch on result Not Equal	Z-Flag = 0
BEQ	Branch on result EQual	Z-Flag = 1

Tabelle 1: Mit bedingten Sprüngen trifft der Prozessor Entscheidungen.

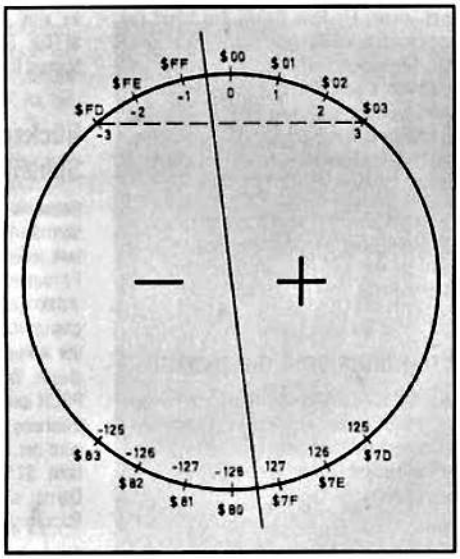


Bild 1: Im Zahlenkreis wird das Prinzip des Zweierkomplementes deutlich: Die komplementären Zahlen liegen einander jeweils gegenüber. Beim Überschreiten der Grenze am unteren Ende liegt ein Überlauf vor.

gebnis eines Ladebefehles oder einer arithmetischen Operation gleich Null ist. Bei anderen Resultaten wird die Zero-Flag wieder gelöscht.
Bit 7, die N-Flagge, zeigt das Vorzeichen des Ergebnisses an. Sie übernimmt bei einem Ladebefehl oder einem ADC- oder SBC-Kommando das höchstwertige Bit des Ergebnisses. Sie ist also bei einem negativen Resultat gesetzt und bei einem positiven gelöscht. Daher auch ihr Name Negative-Flag.
Diese Flaggen werden auch von anderen Befehlen beeinflußt. Wenn diese an der Reihe sind, werden wir auf diese Nebenwirkungen eingehen.

Zukunftsmusik

Bit 3 des Statusregisters ist ein Prozessorinterner Schalter. Mit speziellen Befehlen kann der Programmierer dieses Flag setzen oder löschen und den Prozessor dadurch in den sogenannten Dezimal-Modus schalten. Die Auswirkungen dieses Schalters werden wir in der nächsten Folge ausführlich besprechen.
Mit den Bits 2 und 4 kann man externe Programm-Unterbrechungen (Interrupts) kontrollieren. Die I-Flagge dient dabei als Schalter und teilt dem Prozessor mit, ob solche Interrupts zugelassen sind. Mit der B-Flagge (Break-Flag) kann man feststellen, woher diese Unterbrechung kam. Sie wird immer gesetzt, wenn der Prozessor auf einen BRK-Befehl trifft. Dann springt der Rechner nämlich zu der gleichen Adresse wie bei einem Interrupt. Diese ist – der Prozessor erwartet es so – in den Adressen $FFFE und $FFFF des ROMs verzeichnet. Mehr zu diesen Flags in einer späteren Folge der Assembler-Schule.

Immer diese Entscheidungen

Jetzt wollen wir aber endlich wieder programmieren! Mit den ersten neuen Befehlen, die Sie kennenlernen sollen, lüften wir das Geheimnis, wie man einzelne Flags des Status-Registers prüfen kann. Das funktioniert bei den Bits Carry, Overflow, Zero und Negative. Das Zauberwort heißt Bedingter Sprung, auf gut Englisch Branch. Die Branch-Befehle testen das entsprechende Status-Bit und führen gebenenfalls einen Sprung aus. Ist die Bedingung nicht erfüllt, wird einfach beim nächsten Befehl weitergemacht. Alle Branch-Befehle des 6502 zeigt Tabelle 1.

Als Parameter verlangen diese Befehle natürlich noch das Sprungziel. In unserem Simulator und wenn Sie mit einem Assembler arbeiten, können Sie einfach die Adresse des Zieles oder noch besser ein Label, das an dieser Adresse steht, hinter den Branch-Befehl schreiben. Der Prozessor benutzt aber nicht diese Adresse, sondern die Differenz zwischen dem nächsten Befehl und dem Sprungziel. Diese Umrechnung führt der Assembler für Sie durch.
Dieser berechnete Offset ist ein Ein-Byte-Wert in Zweierkomplement-Form. Bei einem Rücksprung ist er negativ, bei einem Sprung nach vorne positiv. Mit Branches kann man höchstens 128 Bytes rückwärts beziehungsweise 127 Bytes nach vorne springen. Diese Adressierungsart, die es ausschließlich bei den Verzweigungs-Befehlen gibt, heißt Relative Adressierung. Diese Befehle haben eine Instruktionslänge von zwei Bytes.

Fehlerfalle

Nach so viel Erklärungen sollte das erste Beispielprogramm für Sie keine Geheimnisse mehr bergen. Die ersten drei Befehle sind altbekannt. Danach kommt eine Programmverzweigung. Wenn bei der Addition alles glattgegangen ist, wird der Befehl bei $C00F, Label OKAY, angesprungen. Das Ergebnis der Addition wird dann in SUM2 gespeichert, und das Programm ist fertig.
Ist nach der Addition die V-Flagge gesetzt, wird das Ergebnis verworfen. Stattdessen wird $FF nach Flag gespeichert. Dadurch kann auch nach dem Programmende noch ein Fehler festgestellt werden, zur Not durch PEEK(49173).
Sie sollten in diesem Programm mit den Werten für SUM1 und SUM2 experimentieren und auch den ADC-Befehl mal durch SBC ersetzen. Beobachten Sie dabei auch, was mit den Flaggen Carry, Zero und Negative passiert.

XY aufgelöst

Das zweite Programm bringt eine Reihe neuer Befehle. Gleich der erste ist noch unbekannt. Aber keine Angst, so neu ist er auch wieder nicht.

LDX

heißt LoaD X-register. Er funktioniert fast wie LDA, nur daß eben nicht der Akku sondern das Index-X-Register geladen wird. Für das Index-Y-Register gibt es diesen Befehl auch, er heißt

LDY

Beide Befehle gibt es in der Immediate- und der Absolute-Version, beide beeinflussen – wie LDA – die Flaggen Negative und Zero.

Der nächste Befehl im Programm heißt

TXA

und ist die Abkürzung für Transfer X-register to Akku. Er gehört zu einer Vierergruppe von Befehlen, die Prozessor-intern den Datenverkehr zwischen dem Akku und den Index-Registern ermöglichen. Die anderen drei heißen

TYA (Transfer Y to Akku),

TAX (Transfer Akku to X) und

TAY (Transfer Akku to Y).r

Alle vier sind Ein-Byte-Befehle mit der Adressierungsart Implied, sie beeinflussen die N- und die Z-Flagge.

und kurzen Ein-Byte-Befehl den Inhalt des entsprechenden Registers um Eins zu erhöhen (increment) oder zu vermindern (decrement). Damit eignen sich die Index-Register hervorragend als Zähler für Programmschleifen. Für den Akku gibt es solche Befehle (vielleicht INA oder DEA) leider nicht.

Diese vier Befehle setzen je nach Ergebnis die Zero- und die Negative-Flag. Alle vier gibt es nur als impliziert adressierte Ein-Byte-Befehle.

Programmieren mit System

Der letzte unbekannte Befehl in diesem Programm gehört eigentlich noch gar nicht in diese Lektion. Seine genaue Funktionsweise kommt in der nächsten Folge. Hier nur soviel:

JSR

heißt Jump to SubRoutine, zu Deutsch „Springe in ein Unterprogramm". Er funktioniert ähnlich dem BASIC-Befehl GOSUB. Nach der Ausführung der Unterroutine wird das Programm beim nächsten Befehl fortgesetzt.

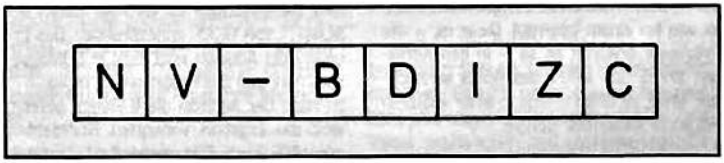


Bild 2: Im Prozessor-Register hat jedes Bit eine besondere Bedeutung.

Den nächsten Befehl wollen wir erstmal übergehen und wenden uns dem Kürzel

INX

zu. Es bedeutet INcrement X-register. Auch dieser Befehl hat drei Geschwister. Sie heißen

INY (INcrement Y-register),

DEX (DEcrement X-register) und

DEY (DEcrement Y-register).

Diese vier Befehle sind unter anderem dafür verantwortlich, daß man die Index-Register X und Y so gerne benutzt. Sie ermöglichen es es nämlich, mit einem schnellen

Das Kürzel PRCH ist ein innerhalb des Simulators vordefiniertes Label und soll an PRint CHaracter erinnern. Das Unterprogramm, zu dem diese Adresse gehört, gibt auf dem Bildschirm das Zeichen aus, dessen ASCII-Code im Akku des Prozessors steht. Im Betriebssystem-ROM des C64 gibt es eine solche Routine an der Adresse $FFD2. Sie ist normalerweise unter dem Namen BSOUT bekannt.

Rücksprünge ergeben Schleifen

Sehen wir uns das Programm nun in Aktion an. Falls Sie es noch nicht getan haben, laden Sie es in den Editor und drücken F7, um es zu assemblieren. Der Befehl LDX #$20 lädt den ASCII-Code des Leerzeichens – das kleinste druckbare Zeichen – ins X-Register. Der nächste Befehl kopiert diesen Wert in den Akku, wo die Routine PRCH ihn erwartet. Nach der Ausgabe des Zeichens (Leerzeichen ist nicht zu sehen) wird der Inhalt des X-Registers um Eins erhöht. $21 ist kleiner als 127, also positiv. Darum ist die N-Flagge gleich Null, und der Rücksprung wird ausgeführt.

Jetzt wird die Schleife erneut durchlaufen und ein Ausrufezeichen ausgegeben. Wenn Sie es nicht sehen, liegt das wahrscheinlich daran, daß Sie im unteren Drittel des Bildschirmes den Monitor eingeschaltet haben. Drücken Sie die HOME-Taste, und Sie sehen an der selben Stelle vier inverse Leerzeilen. Sie stellen in unserem Simulator den Bildschirm dar, auf den die Routine PRCH zugreift.

Sie können nun die Pfeil-aufwärts-Taste drücken, um die Geschwindigkeit zu erhöhen. Nacheinander sollten auf dem 'Bildschirm' die Zeichen bis CHR$(127) gedruckt werden.

Vergleichen Sie bitte

Der letzte neue Befehl dieser Lektion taucht im dritten Beispielprogramm auf und heißt

CMP

(CoMPare to accumulator – vergleiche mit Akku). Er führt eine Subtraktion aus wie der Befehl SBC mit vorher gesetzter Carry-Flagge. Das Ergebnis dieser Subtraktion wird jedoch nicht in den Akku übernommen. Der Befehl CMP dient lediglich dazu, die Flaggen N, Z und C im Status-Register entsprechend zu setzen.

Nach der Ausführung eines CMP-Befehles ist also die Z-Flagge gesetzt, wenn das Ar-

gument und der Inhalt des Akkumulators gleich waren. Wenn die Carry-Flagge gesetzt ist, bedeutet das, daß das Argument größer oder gleich dem Akku-Inhalt ist, gelöschte Carry heißt kleiner.

Der JSR-Befehl tritt in unserem dritten Beispiel mit der Adresse GETC auf. Auch dieses ist ein vordefiniertes Label. Die dazugehörige Betriebssystem-Routine verhält sich ähnlich wie der BASIC-Befehl GET: Bei der Rückkehr aus diesem Unterprogramm enthält der Akku den ASCII-Code der zuletzt gedrückten Taste. Wurde keine Taste gedrückt, so ist die Z-Flagge gesetzt, und im Akku steht eine Null.

Mit Filter

Das Programm fragt die Tastatur ab. Es bricht ab, wenn die RETURN-Taste (ASCII 13) gedrückt wird. Bei anderen Tasten wird überprüft, ob es sich um eine Ziffer handelte. Der Befehl CMP #'0 stellt keine neue Adressierungsart dar, sondern nutzt die Fähigkeit des Assemblers aus, Zeichen in deren ASCII-Wert umzurechnen. Bei der Programmausführung wird dieser Befehl zu CMP #$30, denn $30 (48) ist der ASCII-Code des Zeichens Null.

Die ASCII-Codes in Ihrem C64-Handbuch auf Seite 135ff haben übrigens auch in Maschinensprache Gültigkeit. Der Doppelpunkt ist demnach das nächsthöhere Zeichen nach der 9. Die Befehlsfolge CMP und BCS vergleicht ja auf 'größer oder gleich'. Würde an dieser Stelle CMP #'9 stehen, so würde das Programm die 9 abweisen.

Wenn das eingegebene Zeichen kleiner als 0 oder größer als 9 ist, springt das Programm wieder zur Eingabeschleife. Ziffern werden auf dem Bildschirm ausgegeben.

Sie sollten versuchen, dieses doch schon recht komplizierte Programm in allen Einzelheiten zu verstehen. Es zeigt, wie vielfältig die Branch-Befehle eingesetzt werden können. Dieses Programm dient auch als Grundlage zur „Hausaufgabe" (siehe Programm auf Ihrem Datenträger).

In der nächsten Lektion lernen Sie, wie Sie selbst Unterprogramme schreiben. Außerdem gibt es wieder Beispielprogramme zum Thema Arithmetik. HS

In die Tiefe

INPUT 64-Assemblerschule, Teil 3

In dieser Lektion unseres Maschinensprache-Lehrgangs werden wir uns im doppelten Sinne in neue Tiefen wagen. Zum einen werden Sie den Kellerspeicher kennenlernen. Zum anderen steigen wir von der Ebene der Bytes noch eine Stufe tiefer und schauen uns Befehle an, mit denen einzelne Bits manipuliert werden können. Wenn Sie durch die Beschäftigung mit der Maschinensprache bereits sechzehn Finger haben, werden Sie es vielleicht auch als Abstieg empfinden, daß wir gegen Ende dieses Kapitels wieder im Dezimalsystem rechnen.

In der letzten Folge wurde ja schon der JSR-Befehl vorgestellt. Wir haben gesagt, daß die Abkürzung von 'Jump to SubRoutine' (Springe in ein Unterprogramm)kommt. Hin und wieder liest man auch 'Jump Saving Return-adress' (Springe und rette Rückkehradresse). Diese Interpretation werden Sie verstehen, wenn Sie die genaue Funktionsweise kennengelernt haben. Allerdings müssen wir dazu etwas weiter ausholen.

Der Befehl JSR benutzt den Prozessor-Stack. 'Stack' ist mal wieder ein englisches Wort, übersetzt heißt es Stapel. Dabei handelt es sich um einen besonderen, für diesen Zweck reservierten Speicherbereich. Er liegt bei 6502-Computern an den Adressen $100 bis $1FF, also in der Speicherseite 1.

Man kann sich den Stack anschaulich wie einen Stapel aus Notizzetteln vorstellen: Sie

können auf einen solchen Stapel Zettel oben drauflegen oder von oben jeweils einen wegnehmen. Lesen kann man immer nur den obersten.

Tiefstapler

Der Prozessor-Stack beginnt an der Adresse $1FF und 'wächst' abwärts. Die CPU besitzt zur Verwaltung des Stapels ein eigenes Register, den Stackpointer (Stapelzeiger). In unserem Simulator wird sein Inhalt in der Spalte unter 'SP' angezeigt. Der Stackpointer zeigt immer auf die nächste freie Speicherstelle im Stack. Diesen Mechanismus zeigt Bild 1: Die ersten vier Einträge im Stapel ($1FF bis $1FC) enthalten bereits Informationen. Legt ein Programm nun etwas auf dem Stack ab, so wird die Speicherstelle $1FB beschrieben und gleichzeitig der Stackpointer um Eins vermindert. Beim Lesen aus dem Stack erhöht der Prozessor den Inhalt des Stackpointers um Eins und liest die Speicherstelle $1FC.

Wenn der Prozessor nun auf einen JSR-Befehl trifft, so schreibt er den augenblicklichen Inhalt des Programmzählers auf den Stack, zuerst das höherwertige und dann das niederwertige Byte. Auf diese Art 'merkt' er sich die Adresse des Unterprogrammaufrufes. Dann springt er zu der im Befehl angegebenen Adresse. (Aus Prozessor-internen Gründen wird übrigens die Adresse des dritten Bytes des JSR-Befehles abgespeichert.)

Am Ende eines jeden Unterprogrammes steht der Befehl

RTS

(ReTurn from Subroutine, zu deutsch Rückkehr aus Unterprogramm). Er wird folgendermaßen abgearbeitet: Die CPU holt sich die oberen beiden Bytes vom Stack und lädt sie in den Programmzähler. Der wird dann um Eins erhöht, und an dieser Adresse setzt der Prozessor seine Arbeit fort. An dieser Stelle steht ja genau der nächste Befehl nach dem JSR-Kommando.

Stapellauf

Auf diese Weise kann sich der Prozessor bis zu 128 ineinander verschachtelte Unterprogrammaufrufe merken (jeder belegt zwei Bytes, der Stack kann 256 Bytes aufnehmen). Das erste Beispielprogramm auf Ihrem Datenträger verdeutlicht diesen Mechanismus.

Es beginnt mit einem Sprung in das Unterprogramm INPT, das die Aufgabe hat, die Tastatur abzufragen, die Anzahl der Tastendrücke zu zählen und bei Eingabe der RETURN-Taste abzubrechen. Dazu benutzt es

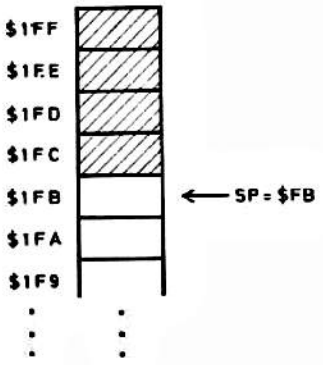


Bild 1: Der Prozessor-Stack wächst abwärts. Die schraffierten Kästchen stellen bereits belegte Speicherstellen dar.

die Routine WAIT. Diese aktiviert in einer Schleife die Betriebssystem-Routine GETC so lange, bis eine Taste gedrückt wird. WAIT gibt den Wert der betätigten Taste im Akku zurück. INPT prüft dann, ob es die RETURN-Taste war. In diesem Falle bricht sie ab und gibt die Anzahl der gedrückten Tasten im X-Register an das aufrufende Programm zurück. Anderenfalls wird das X-Register inkrementiert und die Tastatur erneut abgefragt. Im Hauptprogramm wird nach Beendigung des Unterprogramms INPT die Anzahl der Tasten in der Speicherstelle ZAHL gespeichert und vor dem Programmende noch ein erneuter Tastendruck abgewartet.

Wenn Sie das Programm starten, achten Sie auf den jeweiligen Inhalt des Stackpointers. Mit der Minus-Taste können Sie sich auch den Stack ansehen. Interessant sind hier die Adressen $1F8 bis $1FF. Wie Sie sehen, wird bei der Ausführung des RTS-Befehles im Stack selber nichts geändert. Durch die Inkrementierung des Stackpointers ist die Rückkehradresse jedoch aus der Sicht des Prozessors aus dem Stack entfernt (Achtung: diese Routinen laufen so nicht außerhalb der Assembler-Schulel).

Bestimmt haben Sie die Bedeutung des zweiten Befehles im Beispielprogramm schon erraten, er sei hier aber der Vollständigkeit halber noch erklärt.

STX

heißt STore X-register und speichert den Inhalt des Index-X-Registers an der angegebenen Speicheradresse ab. Wie Sie völlig richtig vermuten, gibt es diesen Befehl auch für das Y-Register, er heißt dann

STY

Schiebung

Wo wir schon mal dabei sind, neue Befehle zu lernen, können wir eigentlich auch gleich damit weitermachen. Die nächsten vier Instruktionen bilden die Gruppe der Schiebe- und Rotationsbefehle. Ihre Funktionsweise ist in Bild 2 dargestellt. Links sehen Sie die beiden Schiebebefehle. Sie heißen

ASL (Arithmetic Shift Left)

und

LSR (Logical Shift Right).

Sie bewirken eine Verschiebung des angesprochenen Bytes um eine Bitstelle nach links (ASL) beziehungsweise nach rechts (LSR). Das dabei freiwerdende Bit wird mit einer Null aufgefüllt, das herausgeschobene Bit landet in der Carry-Flagge. Neben der Carry werden je nach Ergebnis noch die Zero- und die Negativ-Flagge beeinflußt.

Mit den Schiebebefehlen kann man – je nach Adressierungsart – entweder den Akku-Inhalt verändern oder auf eine Speicherstelle zugreifen. Ohne Argument (Adressierungsart Implied) fühlt sich der Akku angesprochen, ansonsten die angege-

bene Adresse. (Allerdings verlangen einige Assembler die Schreibweise 'ASL A', wenn die Implied-Version gemeint ist.)

Diese Befehle werden sehr oft benötigt, wenn irgendwelche Bytes bitweise verarbeitet werden sollen. Außerdem dienen sie zur Multiplikation und Division. Wenn man im Dezimalsystem eine Zahl um eine Stelle nach links verschiebt, ergibt das eine Multiplikation mit Zehn. Genauso entspricht die Verschiebung einer Binärzahl um ein Bit nach links einer Multiplikation mit Zwei. Analog ergibt das Verschieben nach rechts eine Division.

Bitkarussell

Die beiden anderen Operationen dieser Vierergruppe sind die sogenannten Rotationsbefehle. Sie sind den Schiebebefehlen sehr ähnlich. Die Abkürzungen für die Rotationsbefehle sind

ROL (ROtate Left)

und

ROR (ROtate Right).

Auch bei ihnen wird das jeweilige Byte um ein Bit nach links (ROL) oder nach rechts (ROR) verschoben, wobei das überschüssige Bit in die Carry gelangt. Am anderen Ende wird jedoch nicht eine Null nachgeschoben sondern der Inhalt, den die Carry vor der Ausführung des Befehles hat.

Für die Rotationsbefehle gilt in Bezug auf die Beeinflussung der anderen Status-Flaggen und die Adressierungsarten das gleiche wie bei den Schiebebefehlen.

Diese Operationen werden benötigt, wenn der Inhalt der Carry-Flagge von Bedeutung ist. Zum Beispiel kann man eine Zwei-Byte-Zahl mit der Befehlsfolge

ASL Low-Byte

ROL High-Byte

verdoppeln. Das höchste Bit des Low-Bytes landet dabei in Bit 0 des High-Bytes.

Logisch!

Die nächste Gruppe von Befehlen, die Sie kennenlernen sollen, stellen drei verschiedene Operationen zur Verfügung. An ihnen ist – ähnlich wie bei den Befehlen ADC und SBC – der Akku und ein anderes Byte beteiligt. Alle drei kennen dieselben Adressierungsarten wie ADC und SBC, und sie beeinflussen die Negativ- und die Zero-Flagge. Es handelt sich um die logischen Verknüpfungen von Binärzahlen.

Der 6502-Prozessor kennt Maschinenbefehle für die drei Logik-Funktionen Und, Oder und Exklusiv-Oder. Die Wertetabellen der drei Verknüpfungen sind in Tabelle 1 enthalten.

Der zur Und-Verknüpfung gehörende Maschinensprachebefehl heißt

AND (AND with akku)

Mit ihm werden der Akku und das Argument bitweise gemäß der abgebildeten Wertetabelle verknüpft. Das Ergebnis steht danach im Akku.

Mit diesem Befehl lassen sich gezielt Bits im Akku löschen. Alle Bits, die im Argument gleich Null sind, sind es auch im Ergebnis. Ein im Argument gesetztes Bit beeinflußt das entsprechende Akku-Bit nicht. Beispielsweise löscht der Befehl 'AND #$0F' das obere Nibble im Akku. Das niederwertige Nibble enthält nach der Ausführung immer noch seinen alten Wert. Diesen Vorgang nennt man Maskieren, das Byte $0F stellt eine Bit-Maske dar.

Mit dem Befehl

ORA (OR with Akku)

wird eine bitweise Oder-Verknüpfung zwischen dem Akku und dem Argument durchgeführt, dessen Ergebnis wieder im Akku steht. Mit ihm werden einzelne Bits im Akku gesetzt. Alle Eins-Bits im Argument werden als solche in den Akku übernommen.

Um eine bitweise Exklusiv-Oder-Verknüpfung zwischen dem Akku und einem Argument durchzuführen, benutzt man den Befehl

EOR (Exclusive OR with akku)

Auch bei ihm nimmt der Akku das Ergebnis auf. Alle Akku-Bits, die im Argument gesetzt sind, werden durch diesen Befehl invertiert. Mit 'EOR #$FF' kann man so auf einfache Art und Weise das Einerkomplement der im Akku stehenden Zahl berechnen lassen.

Neben dem gezielten Manipulieren einzelner Bits im Akku werden die logischen Verknüpfungen auch zum Testen von Zahlen verwendet. So ist nach einem EOR-Befehl die Z-Flagge genau dann gesetzt, wenn Akku-Inhalt und Argument gleich waren. Im Gegensatz zum SBC-Befehl bleiben die V-Flagge und die Carry jedoch erhalten.

Um die Wirkungsweise dieser drei Befehle zu verstehen, sollten Sie sie mit dem Simulator ausgiebig ausprobieren. Erinnern Sie sich dazu an Ihr allererstes Maschinensprache-Programm zurück.

Begib Dich direkt dorthin

Nach so viel Theorie wollen wir uns wieder mal einem Programm zuwenden. Das zweite Beispiel auf Ihrem Datenträger enthält eine Routine, mit der ein Byte in hexadezi-

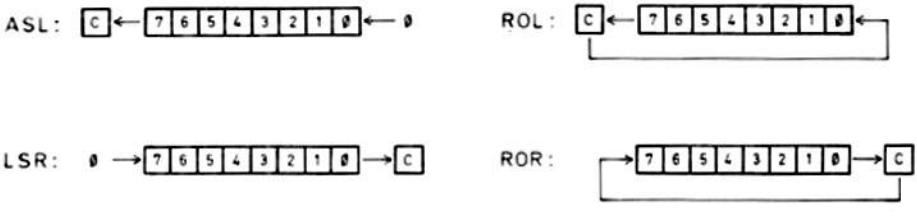


Bild 2: Die Schiebe- und Rotationsbefehle benutzen die Carry-Flagge als achtes Bit.

A	B	A AND B	A OR B	A EXOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabelle 1: Die Funktionsweise der logischen Operationen erkennt man aus ihren Wahrheitstafeln.

maler Darstellung auf dem Bildschirm ausgegeben werden kann.

Warum das Programm mit dem Befehl

JMP

beginnt, steht einige Zeilen tiefer. Aber daß er ausgeschrieben JuMP heißt und 'springe' bedeutet, sollen Sie jetzt schon wissen. Dieser Drei-Byte-Befehl veranlaßt den Prozessor, seine Arbeit an der angegebenen Adresse fortzusetzen, und zwar ohne Rücksicht auf den Inhalt des Status-Registers. Ein weiterer Unterschied zu den Branch-Befehlen ist, daß man mit JMP beliebige Entfernungen innerhalb des Speichers überbrücken kann. Die Zieladresse steht explizit im Befehl und nicht als Differenz zum augenblicklichen Stand des Programmzählers.

Sichtbarer Erfolg

In der nächsten Programmzeile wird ein Label eingeführt, unter dem unsere Routine das auszugebende Byte finden soll.

Die folgende Zeile ist das Sprungziel des JMP-Befehls und der Anfang des eigentlichen Programmes. Die ersten beiden Befehle sorgen dafür, daß ein Dollarzeichen als Kennung für 'Hexadezimal' ausgegeben wird.

Das auszugebende Byte wird alsdann in den Akku geladen und viermal nach rechts geschoben. Dadurch gelangt der Inhalt des oberen Nibbles – das ja zuerst auf dem Bildschirm erscheinen soll – in die unteren vier Bits des Akkus, und das obere Nibble wird gleich Null.

So wird es auch von der Routine PNIB erwartet, die dazu dient, eine Zahl zwischen $00 und $0F auszugeben. Nach ihrer Beendigung lädt das Programm den Inhalt von WERT erneut in den Akku und maskiert mit dem Befehl 'AND #$0F' das untere Nibble aus. So kann es ebenfalls mittels PNIB ausgegeben werden. Zu dem folgenden RTS kommen wir noch.

PNIB soll eine Hex-Ziffer ausgeben. Aus einer ASCII-Tabelle entnehmen wir: Die Ziffern 0 bis 9 haben die Codes $30 bis $39, zu den Buchstaben A bis F gehören die ASCII-Werte $41 bis $46.

Die Routine PNIB arbeitet nun folgendermaßen: Wie bereits gesagt, erwartet sie im Akku eine Zahl mit gelöschtem High-Nibble. Durch den ORA-Befehl werden die unteren vier Bits nicht angetastet (im Argument sind sie gleich Null), und das höherwertige Nibble erhält eine Drei. Für die Ziffern 0 bis 9 enthält der Akku also schon den richtigen Wert zur Übergabe an die Betriebssystem-Routine PRCH. Das wird durch die nächsten beiden Befehle überprüft. Sie bewirken einen Sprung zum Label OKAY, wenn der Akku eine ASCII-Ziffer enthält.

Den Befehl „CMP #'9+1" wollen wir etwas genauer unter die Lupe nehmen: Der Assembler erkennt beim Übersetzen an dem Apostroph, daß ein ASCII-Zeichen folgt und setzt dafür den entsprechenden Wert ein. Die Anweisung „+1" läßt ihn noch Eins addieren. Im Simulator liest sich der Befehl somit als „CMP #$3A", also Vergleich mit dem der 9 folgenden Zeichen. Und so soll es auch sein, weil die CMP-BCC-Sequenz auf 'kleiner' und nicht auf 'kleiner oder gleich' prüft.

Wird der Sprung nicht ausgeführt, so enthält der Akku mindestens $3A. Zu $41, dem Wert für ein A, fehlen also noch 7. Da beim Erreichen des ADC-Befehls die Carry in jedem Falle gesetzt ist, kann man sich sparen, sie zu löschen. Man gibt im Argument einfach einen weniger an. So erklärt sich der Befehl „ADC #6".

Das Folgende ist wieder simpel: Mit „JSR PRCH" wird das Zeichen ausgegeben, und der RTS-Befehl führt in das aufrufende Programm zurück.

Von BASIC nach Maschine

Das besprochene Programm eignet sich sehr gut als kleines Hilfsprogramm, das man auch von BASIC aus aufrufen kann. Dazu überspielen Sie es aus dem Hauptmenue der Assemblerschule auf einen eigenen Datenträger. Dann schalten Sie den Rechner kurz aus und wieder ein, laden den INPUT-Ass (den Assembler von der Rückseite der Diskette) und mit ihm das abgespeicherte Beispielprogramm. Assemblieren Sie es entweder in den Speicher oder, wenn Sie es für später lauffähig aufheben wollen, auf einen Datenträger. Dann können Sie es mit LOAD"name",8,1 einfach wieder laden.

Gestartet wird das assemblierte Programm mit SYS 49152 (49152 ist gleich $C000, der Startadresse). Vorher können Sie mit POKE 49155,zahl den auszugebenden Wert festlegen.

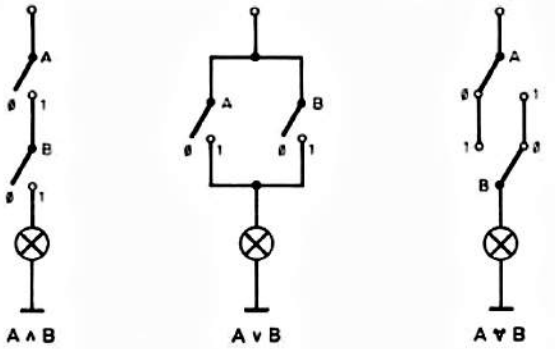


Bild 3: Die logischen Verknüpfungen AND, OR und EXOR lassen sich auch schaltungstechnisch darstellen.

Nun zu dem oben angekündigten tieferen Sinn des JMP-Befehles am Anfang des Programmes: Bislang haben wir Zahlen immer hinter einem Programm aufgehoben. Solange man nur in Assembler programmiert, ist das auch in Ordnung. Bei einem Programm, das man aus BASIC heraus aufrufen will und das Werte übergeben bekommt, ist die hier vorgestellte Reihenfolge jedoch sinnvoller. Wenn Sie nämlich Änderungen an dem Programm vornehmen, etwa um hinter dem Byte noch ein RETURN auszugeben (das sollten Sie übrigens mal probieren!), ändert sich automatisch die Programmlänge, und Sie müßten sich eine andere POKE-Adresse für die Wertübergabe merken.

. . . und wieder zurück

Der erste RTS-Befehl (der an der Adresse $C01B) bewirkt eine Rückkehr nach BASIC. Der BASIC-Interpreter legt nämlich bei der Ausführung des SYS-Befehles automatisch eine Rückkehr-Adresse auf dem Stack ab. Dadurch ist es möglich, auch aus BASIC-Programmen heraus eine Maschinensprache-Routine aufzurufen und anschließend mit dem folgenden BASIC-Befehl weiterzumachen.

Sie können in den bisherigen Beispielprogrammen den abschließenden BRK-Befehl auch durch ein RTS ersetzen, wenn Sie sie im BASIC-Direktmodus ausprobieren wollen. Dadurch vermeiden Sie, daß der Bildschirm nach der Abarbeitung gelöscht wird.

Innerhalb des Kurses liegt übrigens beim Start des Simulators immer die Startadresse des Editors auf dem Stack. Diese können Sie auch unter dem vordefinierten Label EXIT erreichen.

Zehn statt sechzehn

Zum Schluß wollen wir uns noch – wie in der letzten Folge angekündigt – die Wirkung der D-Flagge ansehen. Wenn sie gesetzt ist, arbeitet der Prozessor im sogenannten Dezimal-Modus. Für die Additions- und Subtraktionsbefehle werden dann Zahlen im BCD-Format verwendet. BCD heißt 'Binary Coded Decimal', also binär kodierte Dezimalziffer.

Normalerweise kann man mit einem Nibble sechzehn verschiedene Zahlen darstellen. Im BCD-Code verwendet man nur zehn davon (siehe Tabelle 2), also jeweils ein Nibble für eine Dezimalziffer.

Die gesetzte D-Flagge im Status-Register bewirkt nun, daß das Ergebnis einer Addition oder einer Subtraktion automatisch im BCD-Code dargestellt wird. Ein Überlauf in das nächsthöhere Nibble entsteht nicht

Binär	BCD
%0000	0
%0001	1
%0010	2
%0011	3
%0100	4
%0101	5
%0110	6
%0111	7
%1000	8
%1001	9
%1010	ungültig
%1011	ungültig
%1100	ungültig
%1101	ungültig
%1110	ungültig
%1111	ungültig

Tabelle 2: Der besseren Lesbarkeit und Genauigkeit der BCD-Darstellung fallen sechs Bit-Kombinationen zum Opfer.

erst beim Überschreiten von 15, sondern bereits wenn das Resultat größer als neun wird. Anschauliche Beispiele dazu finden Sie in den Erklärungen auf Ihrem Datenträger.

Der Befehl, mit dem man die D-Flagge setzt, heißt

SED (SEt Decimal-flag)

In den 'normalen' Rechenmodus können Sie den Prozessor mit dem Befehl

CLD (CLear Decimal-flag)

zurückschalten. Diesen Befehl sollten Sie auch zu Anfang eines jeden Programmes ausführen lassen, wenn unklar ist, in welchem Zustand sich der Prozessor beim Programmstart befindet.

Verwandlung

Das dritte Beispielprogramm in dieser Folge benutzt den Dezimal-Modus der CPU zur Umrechnung einer Zwei-Byte-Hexadezimalzahl in eine fünfstellige BCD-Zahl. Das Programm beginnt wieder mit einem Sprung, der die verwendeten Datenspeicher überbrückt.

Das eigentliche Programm beginnt mit dem Löschen des Ergebnis-Puffers. Dann wird das Index-Y-Register als Zähler für sechzehn Schleifendurchläufe – für jedes Bit einen – initialisiert und die Dezimal-Flagge gesetzt.

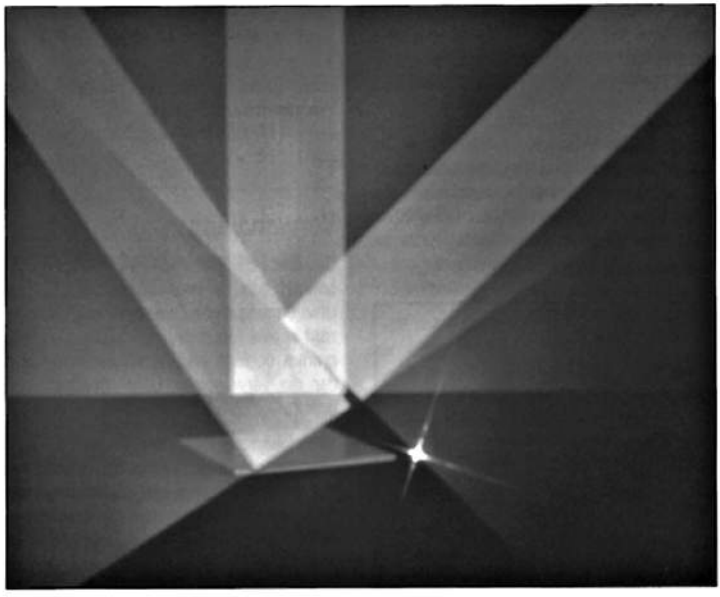
Die ersten beiden Befehle innerhalb der Schleife bewirken eine Verdoppelung der umzuwandelnden Zahl und eine Übertragung des höchstwertigen Bits in die Carry-Flagge.

Die folgenden neun Befehle bewirken beim ersten Schleifendurchlauf noch nicht viel. Jedoch handelt es sich hierbei um die Verdoppelung des Inhaltes des Ergebnisspeichers im BCD-Format mit gleichzeitiger Addition des Überlaufs aus ARGU. Dadurch, daß diese Schleife sechzehnmal durchlaufen wird, wird jedes Bit aus ARGU stellenrichtig zu ERGB addiert.

Nach dem letzten Schleifendurchlauf löscht das Programm die D-Flagge wieder und kehrt zum Aufrufpunkt zurück. Dort könnte beispielsweise eine Routine folgen, die den Inhalt von ERGB als fünfstellig formatierte Dezimalzahl auf dem Bildschirm ausgibt.

Diese Routine sollten Sie übrigens mit dem in dieser Folge Gelernten erstellen können. Das wäre dann auch die „Hausaufgabe". Dazu ist zwar eine ganze Menge Tipparbeit nötig, aber immerhin entwickeln Sie auf diese Weise ein Programm, das – ähnlich wie das zweite Beispiel – einen praktischen Nutzen hat. Wie gewohnt werden wir Ihnen in der nächsten Folge eine Musterlösung vorstellen.

Hajo Schulz



Viele Wege führen ins RAM

INPUT 64-Assembler-Schule, Teil 4

Sie erinnern sich bestimmt an die vier bereits vorgestellten Adressierungsarten: Die meisten Befehle arbeiten „immediate" und „absolute". Einige Befehle brauchen keine Parameter und sind „implied" adressiert, die Branch-Befehle benutzen ausschließlich die relative Adressierung.

Die 6502-CPU kennt aber insgesamt elf Adressierungsarten. Mit den bisher unbekannten werden Sie in der Lage sein, Programme zu schreiben, die auf Parameter und Adressen zugreifen, die nicht fest im Programm stehen, sondern erst während des Programmlaufes berechnet werden und sich ändern können. Die Adressierungsart,

Ging es in der letzten Folge der Assembler-Schule noch darum, einzelne Bytes in ihre Bestandteile zu zerlegen, so werden wir uns in dieser Lektion damit befassen, möglichst viele Bytes mit einem Befehl in den Griff zu bekommen. Wir zeigen ihnen, wie mit neuen Adressierungsarten die bekannten Befehle besser und flexibler einzusetzen sind.

die wir zuerst betrachten wollen, stellt diese Möglichkeiten zwar nicht zur Verfügung, ermöglicht aber schnellere und kürzere Programme als bisher gewohnt.

Seite Null

Wie bereits in einer früheren Folge der Assembler-Schule erwähnt, ist der Speicher eines jeden 6502-Rechners in 256 Seiten zu je 256 Bytes eingeteilt. In der letzten Lektion haben Sie gesehen, daß die Seite eins den Platz für den Prozessor-Stack zur Verfügung stellt. Auch die Seite Null (die Zero-Page), das sind die Adressen 0 bis 255 ($00 bis $FF), hat für den Prozessor eine besondere Bedeutung. Auf sie kann nämlich mit der sogenannten direkten oder Zero-Page-Adressierung zugegriffen werden.

Im Prinzip funktioniert diese Adressierungsart genau wie die absolute Adressierung. Der Unterschied besteht darin, daß zur Angabe einer Zero-Page-Adresse nur ein Byte benötigt wird; alle anderen Adressen sind ja bekanntlich Zwei-Byte-Werte. Zu allen Befehlen, die mit der Adressierungsart „absolute" arbeiten können (bis auf den JMP-Befehl), gibt es auch eine Zero-Page-Version. In dieser Variante handelt es sich dann um Zwei-Byte-Befehle, die nicht nur um ein Byte kürzer als die gewohnten Befehle sind, sondern auch schneller abgearbeitet werden.

Im Programmtext unterscheiden sich die beiden Versionen dieser Befehle nicht. Der Assembler erkennt, daß die angegebene Adresse auf der Zero-Page liegt, und fügt bei der Übersetzung automatisch den kürzeren Maschinenbefehl ein. Die CPU erkennt bei der Abarbeitung des Programmes bekanntlich am ersten Byte eines Befehles die Instruktionslänge. So ist auch die Zero-Page-Adressierung wie alle anderen Adressierungsarten im Instruktions-Byte verschlüsselt.

Kehrseite

Die Adressen 0 und 1 der Zero-Page sind beim C64 nicht verfügbar. Das liegt daran, daß er mit einer 6510-CPU ausgestattet ist, die diese beiden Register für besondere Zwecke benutzt. Hier ist der gravierendste Unterschied zum 6502, der ja der „Vater" dieses Prozessors ist. Im C64 wird mit den Registern 0 und 1 die Speicherkonfiguration eingestellt. Dazu mehr in der nächsten Folge.

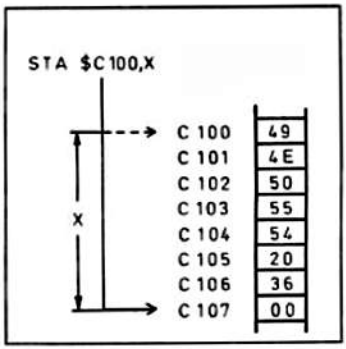


Bild 1: Bei der absolut indizierten Adressierung bildet das Index-Register den Offset.

Da es sich bei der Zero-Page um einen „schnellen" Speicherbereich handelt und – wie wir noch sehen werden – die mächtigsten Adressierungsarten Platz auf der Zero-Page benötigen, werden von den 254 verfügbaren Speicherstellen die meisten vom Betriebssystem des C64 benutzt. Wenn man auf diesen Speicherbereich zugreift, muß man sich im klaren darüber sein, daß man dadurch das Betriebssystem durcheinanderbringen kann und eventuell einen Absturz des Rechners riskiert.

Unbenutzt sind in der Zero-Page nur die Adressen 2 sowie $FB bis $FE. Innerhalb unseres Simulators stehen Ihnen die Adressen $F8 bis $FF zur freien Verfügung.

Sie haben in der letzten Folge der Assembler-Schule gelernt, wie man Speicherstellen, die ein Programm zur Parameterübergabe benötigt, an den Anfang des Programmes legt. Dadurch bleiben die Adressen auch bei einer Programmänderung konstant. Eine andere beliebte Methode, Werte an ein Programm zu übergeben, besteht darin, die Zero-Page dafür zu benutzen. Dann kann das Programm auch in einen anderen Speicherbereich assembliert werden, ohne daß sich diese Adressen ändern. Auch die Rückgabe von Ergebnissen eines Programmes erfolgt häufig über die Zero-Page.

In unserem ersten Beispielprogramm benutzen wir eine Speicherstelle der Zero-Page als Zwischenspeicher. Bevor wir diese Routine unter die Lupe nehmen, sollen Sie noch eine andere Adressierungsart kennenlernen, die dort verwendet wird. Dabei werden die Indexregister X und Y erstmals als solche benutzt. Bisher haben wir sie ja nur als Zähler eingesetzt.

Adressen plus Index

Die Adressierungsart, um die es geht, heißt „Absolut Indiziert". Bild 1 zeigt das Prinzip an einem Beispiel. Dabei enthält das X-Register eine 7. Diese Adressierungsart wird mit folgender Schreibweise gekennzeichnet:

STA $C100,X

Zu der im Befehl angegebenen Adresse wird dabei noch der derzeitige Inhalt eines der Indexregister addiert. Das Resultat dieser Addition ergibt dann die Adresse, auf den der Befehl zugreift.

Diese Adressierungsart funktioniert bei den meisten Befehlen, die mit der absoluten Adressierung arbeiten können. Eine genaue Aufstellung der Befehle mit den möglichen Adressierungsarten bringen wir in der nächsten Folge der Assembler-Schule. Sie ist auch in den Büchern enthalten, die Sie am Ende dieses Artikels aufgelistet finden. Innerhalb unseres Simulators können Sie nach dem Motto „Probieren geht über Studieren" arbeiten. Eine Adressierungsart, die für den eingegebenen Befehl nicht existiert, wird schon bei der Programmeingabe abgewiesen.

Bei den meisten Befehlen können Sie auch das Y-Register als Index verwenden. Die Notation lautet dann beispielsweise

CMP $C123,Y

Diese Adressierungsarten finden immer dann Verwendung, wenn in einem Speicherbereich mit jedem Byte die gleiche Operation durchgeführt werden soll. Mit ein und demselben Befehl ist es – je nach dem laufenden Inhalt des entsprechenden Index-Registers – möglich, auf 256 aufeinanderfolgende Bytes zuzugreifen. Damit lassen sich beispielsweise Tabellen bis zu dieser Länge sehr einfach verwalten.

Sie können zum Beispiel eine Schleife programmieren, die bei jedem Durchlauf das X-Register inkrementiert und dadurch jedesmal eine andere Adresse anspricht. Listing 1 zeigt eine solche Routine. Mit wenigen Befehlen kopiert sie einen Speicherbereich von 256 Bytes Länge in einen anderen.

Wenn Sie sich nun das erste Beispielprogramm auf Ihrem Datenträger anschauen, werden Sie wahrscheinlich nicht auf Anhieb die indiziert adressierten Befehle finden. Sie werden zunächst über einige bisher gänzlich unbekannte Befehle stolpern.

Namen statt Zahlen

Dabei handelt es sich nicht um neue Maschinensprache-Befehle. Es sind vielmehr Anweisungen an den Assembler. Mit ihnen werden Labels, also Assembler-Variablen, deklariert. Bislang haben wir Labels immer nur für Adressen innerhalb des Programmes benutzt. Aber ein Assembler kann sich unter einem Label-Namen beliebige Zahlen und auch Adressen außerhalb des Programmes merken und bei der Übersetzung einsetzen.

Der erste Befehl weist der Variablen RTRN die Zahl 13, das ist der ASCII-Wert der RETURN-Taste, zu. Dadurch kann überall im

```
C083:   ;...
C083:   ;256 BYTES VON ORIG
C083:   ;NACH COPY KOPIEREN
C083:          LDX  #0
C085:   LOOP   LDA  ORIG,X
C088:          STA  COPY,X
C08B:          INX
C08C:          BNE  LOOP
C08E:   ;...
```

Listing 1:
Wenn Sie diese Routine abtippen und in eigenen Programmen einsetzen, müssen die Labels ORIG und COPY vorher deklariert sein.

Programm, wo dieser Wert gebraucht wird, statt dessen der Name verwendet werden. Durch diese Maßnahme werden Programme besser lesbar. „Im richtigen Leben" kann man sich auch Label-Tabellen erstellen, die je nach Bedarf beim Programmieren in den Text geladen werden. Sie können sich auf diese Weise eine Menge Tipparbeit sparen.

Die nächsten vier Befehle haben denselben Sinn: DELC erhält den Wert der DEL-Taste, LEER den der Leertaste, LEFT steht im Programm für Cursor-Links, und unter CRSR steht ein Grafikzeichen zur Verfügung, das in der Routine als Cursor dient.

Bei dem vorgestellten Programm handelt es sich um eine Routine, die Sie auch in eigene Programme einfügen können. Sie enthält einen simplen Zeilen-Editor. Man kann mit ihr bis zu 255 Zeichen von der Tastatur holen und in einem Speicherbereich ablegen. Das jeweils zuletzt eingegebene Zeichen ist mit der DEL-Taste zu löschen.

Die Zero-Page-Adresse $FE wird durch die folgende Anweisung mit dem Label MERK versehen. Sie wird als schneller Zwischenspeicher für das X-Register dienen.

Der Beginn des Bereiches, in dem der eingegebene String gespeichert werden soll, ist im Programm durch das Label BUFF gekennzeichnet. Das bewirkt die Zeile

```
BUFF = $C100
```

Dadurch, daß auch diese Adresse in einer Assembler-Variablen gehalten wird, brauchen Sie später bei Bedarf nur diese Zeile zu ändern, und die Routine benutzt einen anderen Bereich als Puffer.

INPUT selbstgemacht

Der erste eigentliche Befehl – unter der Überschrift – initialisiert das Index-X-Register. Sein Inhalt wird dann für später in die Speicherstelle MERK gerettet. Beachten Sie die Instruktionslänge von nur zwei Bytes – MERK liegt auf der Zero-Page.

Im nächsten Befehl sehen Sie, wie auf die zu Anfang vereinbarten Namen zugegriffen wird: Der Assembler setzt bei der Übersetzung für CRSR wieder 164 ein, und im Trace-Protokoll wird der Befehl als LDA #$A4 auftauchen.

Ab EIN1 wird die Tastatur abgefragt; diesen Mechanismus haben Sie ja in der letzten Folge schon kennengelernt. Ist ein Tastendruck erfolgt, so wird das X-Register wieder mit seinem alten Wert geladen. Die Routine GETC könnte es verändert haben, da sie nur innerhalb der Assemblerschule das X-Register unberührt läßt.

Falls es sich bei dem eingegebenen Zeichen um ein RETURN handelt, verzweigt das Programm nach RAUS, die DEL-Taste bewirkt einen Sprung nach DELT. Andere Sonderzeichen werden durch die Befehle $C01D bis $C023 abgewiesen. Der folgende Befehl gibt das empfangene Zeichen als Echo auf dem Bildschirm aus.

Jetzt kommt das eigentlich Spannende an diesem Programm: Der Befehl

```
STA BUFF,X
```

speichert das empfangene Zeichen ab. Durch die Verwendung der X-indizierten Adressierung und den nachfolgenden INX-Befehl wird das erste Zeichen an der Adresse $C100 abgespeichert, das nächste bei $C101 und so weiter. Ein Speicherbereich, der mit der indizierten Adressierung erreicht werden soll, muß übrigens nicht immer auf der ersten Adresse einer Seite beginnen; die Basis-Adresse ist beliebig.

Der folgende Branch-Befehl wacht darüber, daß nicht mehr als 255 Zeichen eingegeben werden. Wenn nämlich der INX-Befehl eine Null ergibt, würde das nächste Zeichen direkt auf der Speicherstelle BUFF abgespeichert werden. Der Sprung nach DEL1 bewirkt, daß das 256. eingegebene Zeichen sofort wieder gelöscht wird.

Ist das Ende des Puffers noch nicht erreicht, so wird der folgende Branch-Befehl wirksam, und das Programm wartet auf die nächste Taste.

Das Label RAUS ist der Anfang vom Ende der Routine: Zuerst wird das Ende des eingegebenen Strings mit einem Null-Byte markiert. Für diese Markierung könnte man auch einen anderen Wert benutzen, die Null hat sich aber nun mal eingebürgert. Der BASIC-Interpreter macht es genauso, und das zweite Programm – das der Textausgabe dienen wird – verwendet dieses Kennzeichen auch.

Die folgenden Befehle entfernen den Cursor vom Bildschirm, springen in die nächste Textzeile, und der RTS-Befehl schließlich übergibt die Kontrolle wieder dem aufrufenden Programm. Ihn können Sie zu Testzwecken innerhalb des Kurses durch ein BRK ersetzen, dann bleiben Sie im Simulator und können noch Speicherbereiche inspizieren.

Indexvergleiche

Der Rest des Programmes dient dem Löschen des letzten Zeichens. Der Befehl

```
CPX
```

ist neu. Er bedeutet ComPare to X-register und bewirkt dasselbe wie der in Lektion 2 beschriebene CMP-Befehl. Der Unterschied besteht darin, daß das Argument nicht mit dem Inhalt des Akkumulators, sondern mit dem X-Register verglichen wird. Mit Ihrer Vermutung, daß es auch einen Befehl namens CPY gibt, haben Sie übrigens recht.

Mit dem CPX-Befehl überprüft unser Programm vor dem Löschen eines Zeichens, ob überhaupt schon eins eingegeben wurde.

Die restlichen Befehle bewirken neben der Cursor-Bewegung eine Dekrementierung des X-Registers. Dadurch wird zwar das letzte Zeichen nicht wirklich aus dem Speicher gelöscht. Die nächste Eingabe wird es jedoch überschreiben.

Kurz und gut

Der Vollständigkeit halber sei hier erwähnt, daß auch auf die Zero-Page indiziert zugegriffen werden kann. Alle Befehle, die mit der Adressierung „absolut,X" funktionieren, können auch in der Version „Zero-Page,X" eingesetzt werden. Ferner kennt die 6502-CPU noch die Adressierungsart „Zero-Page,Y". Sie ist allerdings den Befehlen LDX und STX vorbehalten.

Wegen des bereits erwähnten Gedränges, das auf der Seite Null herrscht, wird diese Möglichkeit des Prozessors beim C64 verhältnismäßig selten genutzt. Größere Tabellen werden normalerweise nicht in der Zero-Page aufbewahrt. Bei kleineren 6502-Systemen wie dem EPAC65 unserer Schwesterzeitschrift c't, die ohne aufwen-

diges Betriebssystem auskommen, sind diese Adressierungen jedoch wegen der Platz- und Zeitersparnis durchaus üblich.

Mit Klammer

Die mächtigsten Adressierungsarten, die die 6502-CPU zur Verfügung stellt, imponieren schon durch ihre Namen: Sie heißen „indiziert indirekt" und „indirekt indiziert". Man spricht auch von Vor- beziehungsweise Nach-Indizierung. Beiden ist gemeinsam, daß sie Zeiger (neudeutsch Pointer) auf der Zero-Page benutzen. Dazu kommt noch jeweils ein Index-Register. Alle Befehle, die diese Adressierungsarten benutzen, sind zwei Bytes lang (Instruktionsbyte und Zero-Page-Adresse).

Ein Zeiger besteht aus zwei aufeinanderfolgenden Bytes, die eine Adresse enthalten. Dabei steht der niederwertige Teil der Adresse (das Low-Byte) im ersten, der höherwertige (High-Byte) im zweiten Byte. Diese Reihenfolge – Sie kennen sie ja schon von der Assembler-Anweisung W – ist beim 6502 überall anzutreffen. (Beispielsweise wird auch die Adresse bei einem Drei-Byte-Befehl in dieser Reihenfolge erwartet – aber darum kümmert sich der Assembler.)

Die Adressierungsart „indiziert indirekt" benutzt zusätzlich zu dem Pointer auf der Zero-Page noch das X-Register. Die Funktionsweise dieser Adressierung soll die linke Hälfte von Bild 2 erklären. Wir setzen voraus, daß das X-Register den Wert 2 enthält. Die Schreibweise lautet

```
LDA ($F8,X)
```

Der Prozessor addiert zunächst den Inhalt des X-Registers zu der angegebenen Adresse. Die beiden an der dadurch angegebenen und der folgenden Zero-Page-Adresse stehenden Bytes werden als Pointer auf die Adresse aufgefaßt, auf die die CPU schließlich zugreift.

Mit dieser Adressierung kann man auf der Zero-Page Adreßtabellen anlegen, auf die sehr leicht zugegriffen werden kann. Wegen der Überfüllung auf der Seite Null ist beim C64 auch diese Adressierungsart eher selten.

Wesentlich bedeutender ist die nach-indizierte Adressierung. Sie benutzt das Index-Y-Register. Auf der rechten Seite von Bild 2 ist dargestellt, wie sie wirkt. Das Y-Register enthält in unserem Beispiel den Wert 3.

Die CPU holt sich aus der angegebenen und der folgenden Adresse einen Pointer. Zu diesem addiert sie den Inhalt des Y-Registers und bildet so die anzusprechende Adresse.

Diese Adressierungsart hat mehrere entscheidende Vorteile: Mit nur einem Zero-Page-Pointer kann durch unterschiedlichen Inhalt des Y-Registers ein Bereich von 256 Bytes adressiert werden. Der Beginn dieses Bereiches kann innerhalb des Programmes berechnet werden und ist nicht unbedingt vom Programmierer fest vorzugeben.

In der Routine PRNT des zweiten Beispielprogrammes wird von dieser Adressierungsart Gebrauch gemacht. Sie dient dazu, einen String mit bekannter Anfangsadresse auf dem Bildschirm auszugeben.

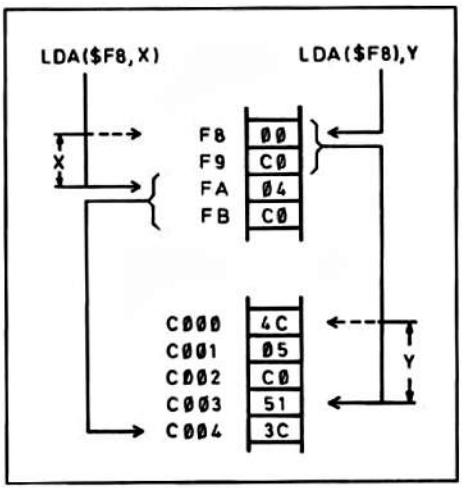


Bild 2: Die mächtigsten Adressierungsarten des 6502 benutzen eine Kombination von Zero-Page-Pointer und Index-Register zur Adreßberechnung.

Jede Menge Labels

Das Programm beginnt mit zwei Label-Zuweisungen. PTRL und PTRH sind zwei aufeinanderfolgende Adressen auf der Zero-Page; sie werden als Pointer auf den auszugebenden Text dienen. ANZT ist eine reine Assembler-Variable und enthält die Anzahl der verschiedenen möglichen Texte.

Schauen wir uns noch die letzten Programmzeilen an, bevor das eigentliche Programm auseinandergenommen wird. Hier stehen die verschiedenen Texte, die das Programm ausgeben kann. Der Befehl B ist eine Assembler-Anweisung. Wir haben sie schon öfter benutzt, um bestimmte Speicherstellen mit einem Wert vorzubesetzen. Trifft der Assembler bei der Übersetzung hinter der B-Anweisung auf Texte in Anführungszeichen, so setzt er die entsprechenden ASCII-Werte in den Programmspeicher ein.

In den Tabellen TABL und TABH sind die nieder- beziehungsweise die höherwertigen Bytes der Anfangsadressen der Texte gespeichert. Ein Kleiner-Zeichen vor einem Argument signalisiert dem Assembler, daß er das niederwertige Byte des folgenden Wertes erzeugen soll, ein Größer-Zeichen ergibt das höherwertige Byte.

Das Label TXTN haben wir noch vergessen; hier ist die Nummer des auszugebenden Textes gespeichert. Wenn Sie andere Texte einsetzen und ein Programm schreiben, das diese Nummer irgendwie berechnet, können Sie das vorliegende Programm zur Ausgabe beliebiger Meldungen verwenden.

PRINT Marke Eigenbau

Das eigentliche Programm beginnt damit, daß es die Nummer des Textes ins X-Regi-

ster lädt, und prüft, ob ein solcher Text überhaupt vorhanden ist. Falls ja, wird das X-Register als Index auf die beiden Text-Tabellen benutzt und die Anfangsadresse des Textes in den Akku und das Y-Register geladen.

Werfen wir nun einen Blick auf die Routine PRNT. Sie erwartet die Anfangsadresse des auszugebenden Textes im Akku und im Y-Register, und zwar das Low-Byte im Akku und das High-Byte im Y-Register. Diese Adresse wird als Pointer in der Zero-Page abgespeichert. Unter Benutzung der indirekt-indizierten Adressierung wird der String nun Byte für Byte ausgelesen und an die Betriebssystem-Routine PRCH übergeben. Diese Routine arbeitet so lange, bis sie auf eine Null trifft oder 256 Zeichen ausgegeben sind.

Zurück zur Adresse $C01D: Hier landet das Programm, wenn das Byte TXTN einen ungültigen Wert enthält. Dann wird die Adresse des Textes ERRT auf die bekannte Art und Weise an PRNT übergeben.

Abschließend gibt das Programm in jedem Falle noch ein Ausrufezeichen aus und wartet auf einen Tastendruck, bevor es zur Aufruf-Adresse zurückkehrt.

Auf eigenen Füßen

Dieses Programm bietet mannigfaltige Möglichkeiten zum Experimentieren. Das einfachste ist, die Zahl bei TXTN zu ändern und dadurch einen anderen der vorhandenen Texte ausgeben zu lassen. Sie können aber auch die Texte selbst oder die Anzahl der Texte ändern. Wenn Sie das Programm in Eigenentwicklungen integrieren wollen, sollten Sie die Befehle, die abschließend auf einen Tastendruck warten, entfernen. Bestimmt fällt Ihnen auch eine Anwendung ein, bei der Sie die beiden in dieser Folge vorgestellten Programme zu einem verbinden.

In der nächsten Folge werden wir die noch fehlenden Maschinensprache-Befehle vorstellen und eine komplette Aufstellung der Befehle und der möglichen Adressierungsarten bringen. Außerdem werden wir einige C64-Besonderheiten besprechen.

Hajo Schulz

Stapelweise Befehle

INPUT 64-Assembler-Schule, Teil 5

Wenn Sie die bisherigen Folgen dieses Kurses aufmerksam durchgearbeitet haben, sind Sie in Maschinensprache schon einigermaßen fit. Die wenigen Befehle, die Sie noch nicht kennengelernt haben, werden diesmal besprochen. Außerdem wollen wir einen Blick auf Eigenheiten des C64 werfen.

Wenn Sie sich die Opcode-Tabelle auf Seite 48 ansehen, werden Sie feststellen, daß Sie die meisten Befehle schon kennen. Wir haben zwar nicht jeden Befehl mit jeder Adressierungsart verwendet. Die Arbeitsweise der Befehle sollte Ihnen aber trotzdem bekannt sein. Es ist übrigens ziemlich überflüssig, diese Tabelle auswendig zu lernen, Sie sollten aber wissen, welche Befehle mit welchen Adressierungsarten kombinierbar sind.

Faulenzer

Der einfachste unter den Befehlen, die wir noch nicht vorgestellt haben, heißt

NOP

Er benötigt keine Argumente und tut auch nichts (NOP heißt No OPeration – keine Operation). Auf den ersten Blick scheint es überflüssig, einen solchen Befehl zur Verfügung zu haben. Aber auch das Nichtstun kostet Zeit. Und manchmal kommt es eben darauf an, die CPU ein wenig warten zu lassen, ohne Registerinhalte zu verändern. Der Befehl findet häufig in zeitkritischen Routinen Anwendung, zum Beispiel bei Programmen zur Bedienung des Kassettenrecorders oder der seriellen Schnittstelle.

Bit-Tester

Ebenfalls bei Programmen, die für Peripherie zuständig sind, wird der Befehl

BIT

gern verwendet. Mit ihm kann abgefragt werden, ob bestimmte Bits in der angesprochenen Speicherstelle gesetzt sind. Er führt eine bitweise AND-Verknüpfung zwischen dem Akku und dem Inhalt der Adresse durch. Im Gegensatz zum AND-Befehl wird das Resultat jedoch nicht in den Akku übernommen. Sein Inhalt bleibt unverändert – etwas Ähnliches kennen Sie ja schon von dem Befehl CMP.

Je nach dem Ergebnis der AND-Verknüpfung wird die Z-Flagge im Statusregister gesetzt. Die Flags N und V enthalten nach der Abarbeitung des BIT-Befehls den Inhalt der Bits 7 beziehungsweise 6 der getesteten Adresse.

Mit diesem Befehl kann man also unabhängig vom Akku-Inhalt die beiden obersten Bits eines Speicher-Bytes testen. Will man andere Bits überprüfen, so muß der Akku vorher mit einer entsprechenden Maske geladen werden.

Speicher als Zähler

Auch die beiden Befehle, die jetzt zur Sprache kommen sollen, sind nicht schwer zu verstehen; sie arbeiten ähnlich wie schon bekannte Kommandos. Gemeint sind

INC (INCrement memory) und
DEC (DECrement memory).

Der Befehl INC tut dasselbe wie INX oder INY, nur wird nicht ein Indexregister, sondern eine Speicherstelle um eins erhöht. Die Flaggen des Statusregisters verhalten sich dabei wie gehabt.

Der DEC-Befehl vermindert den Inhalt der angesprochenen Adresse um eins, auch er beeinflußt die N- und die Z-Flagge wie die Befehle DEX und DEY.

Diese beiden Befehle werden benutzt, wenn ein Schleifenzähler benötigt wird, aber die Register X und Y als Indexregister verwendet werden müssen. Eine andere Anwendung dieser Befehle zeigt Bild 1. Hier wird mit dem INC-Befehl Zeit und Speicherplatz gespart. Bei der Subtraktion einer Ein-Byte-Zahl von einem Zwei-Byte-Wert können Sie mit dem DEC-Befehl den gleichen Trick anwenden.

Datenkeller

In der vorletzten Folge haben Sie den Prozessor-Stack, auch Kellerspeicher genannt, kennengelernt. Sie kennen seine Bedeutung für die Aufbewahrung von Rückkehradressen bei Unterprogrammaufrufen. Das ist aber nur die halbe Wahrheit. Man kann den Stack nämlich auch selbst für die Zwischenspeicherung beliebiger Daten nutzen. Dazu gibt es das Befehlspaar

PHA (PusH Accu) und
PLA (PulL Accu).

Der erste dieser beiden Befehle „schiebt" den Inhalt des Akkus auf den Stapel. Dabei wird der Stackpointer um eins vermindert, diese Operation belegt also nur ein Byte im Stack. (Beim JSR-Befehl sind es jedesmal zwei Bytes.)

Durch den PLA-Befehl wird der so zwischengespeicherte Wert wieder in den Akku zurückgeholt und der belegte Speicherplatz im Stapel freigegeben.

Diese beiden Befehle sind mit äußerster Vorsicht zu genießen. Vergißt man nämlich, einen gePUSHten Wert zurückzuholen, oder läßt den Prozessor einen überflüssigen PLA-Befehl ausführen, so befindet sich der Stack in einem nicht ordnungsgemäßen Zustand – und das nächste RTS kommt bestimmt. Der Prozessor kann ja nicht wissen, ob die beiden oben auf dem Stack liegenden Bytes durch einen PHA- oder durch einen JSR-Befehl dort hingekommen sind. Er interpretiert sie bei einem RTS immer als Return-Adresse. Und daß dieser Sprung „in die Wüste" geht, wenn ein Byte zuviel oder zuwenig auf dem Stack liegt, dürfte wohl klar sein. Diesem Programmierfehler ist besonders schwer beizukommen, da der Prozessor dabei jedesmal woanders landen kann und sich unter Umständen völlig chaotisch verhält.

Leider gibt es die Möglichkeit der Ablage auf dem Stack nicht für die Indexregister. Braucht man sie doch einmal, muß man sich mit einer TXA-PHA- beziehungsweise PLA-TAX-Folge behelfen. (Achtung, dabei geht der Akku-Inhalt verloren!)

Zum Trost haben die Entwickler des 6502 die Möglichkeit vorgesehen, den Inhalt des Prozessor-Status-Registers auf dem Stack abzulegen und wiederherzustellen. Dazu gibt es die Befehle

PHP (PusH Processor status) und
PLP (PulL Processor status).

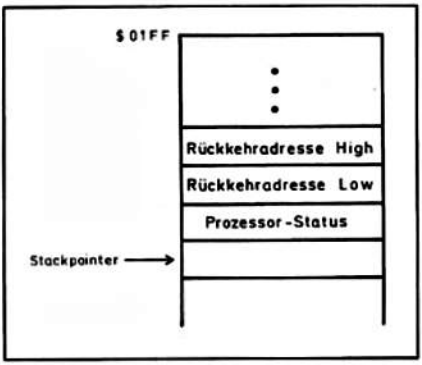
Auch diese beiden Befehle sind nicht ganz ungefährlich. Zusätzlich zu den bei PHA und PLA besprochenen Problemen kann ein unüberlegtes PLP-Kommando beispielsweise dazu führen, daß plötzlich die D-Flagge gesetzt ist und keiner weiß, warum das Programm nur noch Müll berechnet.

```
C018: ;PNTR=PNTR+7
C018:       CLC
C019:       LDA PNTR
C01C:       ADC #7
C01E:       STA PNTR
C021:       LDA PNTR+1
C024:       ADC #0
C026:       STA PNTR+1
C029:       ...
```

```
C018: ;PNTR=PNTR+7
C018:       CLC
C019:       LDA PNTR
C01C:       ADC #7
C01E:       STA PNTR
C021:       BCC OKAY
C023:       INC PNTR+1
C026: OKAY  ...
```

Bild 1: Mit dem INC-Befehl wird die Addition eines Bytes zu einem Zwei-Byte-Wert schneller und kürzer.

Bei einer Unterbrechung wird zuerst die Rückkehradresse, dann das Statusregister auf den Stack gerettet.



Andererseits sind diese beiden Befehle die einzigen, mit denen man direkt an das Statusregister herankommt. Um zu testen, ob die D-Flagge gesetzt ist, muß man zum Beispiel die Befehlsfolge

```
PHP
PLA
AND #%00001000
BEQ HEXA
BNE DEZI
```

verwenden. Es gibt nämlich keinen BDS- („Branch on Decimal Set"-) Befehl.

Das Unterste zuoberst kehren

Wo wir gerade beim Stack sind, sollen hier noch zwei Befehle vorgestellt werden, die auch mit dem Keller zu tun haben. Sie heißen

TSX (Transfer Stack pointer to X) und
TXS (Transfer X to Stack pointer).

Letzterer dient dazu, den Stackpointer zu initialisieren. Das muß zum Beispiel beim Einschalten des Rechners oder nach einem Reset geschehen. Dazu wird das X-Register normalerweise mit $FF geladen und dann der TXS-Befehl ausgeführt. Dadurch sind aus der Sicht des Prozessors alle Eintragungen aus dem Stack entfernt, und es stehen wieder 256 freie Plätze zur Verfügung.

Der TSX-Befehl ist die einzige Möglichkeit, den Inhalt des Stackpointers zugänglich zu machen. Der BASIC-Interpreter prüft zum Beispiel vor jedem GOSUB-Befehl mit einer TSX-CPX-Folge, ob noch genug Platz auf dem Stapel frei ist.

Mit diesem Befehl kann man auch die Kellerstruktur überlisten und zum Beispiel die drittletzte Eintragung aus dem Stack lesen:

```
TSX
LDA $103,X
```

Dabei wird weder der Stackpointer noch der Inhalt des Stapels selbst beeinflußt.

Anlasser

Alle Befehle und Programme, die wir bislang besprochen haben, arbeiten genau voraussehbar. Das heißt, wenn man die Registerinhalte des Prozessors und das Programm kennt, kann man genau voraussagen, welche Aktionen in welcher Reihenfolge durchgeführt werden. Es gibt aber auch Möglichkeiten, diese Abarbeitung eines Programmes von außen zu beeinflussen. Dazu gibt es die sogenannten Unterbrechungen oder Interrupts.

Beim 6502 gibt es drei Arten, von außen eine Programmunterbrechung herbeizuführen. Eine davon ist der Reset. Der Prozessor hat dafür eine spezielle Leitung. Wird an dieser ein Impuls angelegt, so bricht der Prozessor sofort seine Arbeit ab. Er lädt dann den Programmzähler mit dem Inhalt der Adressen $FFFC und $FFFD. Diese Adressen liegen im ROM. Das ist bei jedem 6502-Rechner so, denn auch beim Einschalten wird der Programmzähler mit dieser Adresse geladen, und die soll ja jedesmal denselben Wert enthalten, damit der Computer in einen definierten Ausgangszustand kommt.

Das Laden des Programmzählers mit der Reset-Adresse bewirkt nichts anderes als einen Sprung zu dieser Adresse. Beim C64 ist es die Adresse $FCE2 (64738). An dieser Stelle (auch noch im ROM) steht eine Routine, die zunächst den Stackpointer initialisiert, die D-Flagge löscht und überprüft, ob ein Autostart-Modul im Extension-Port steckt. Falls nicht, werden als nächstes die Peripherie-Chips initialisiert. Dazu gehört zum Beispiel, daß Hintergrund- und Rahmenfarbe gesetzt werden, die Lautstärke des SID auf null geht und die Schnittstellen (Userport, serieller Bus, Kassette etc.) in einen definierten Zustand versetzt werden.

Dann werden diverse Pointer und Vektoren gesetzt. (Zu dem Begriff Vektoren kommen wir in wenigen Augenblicken.) Zum Schluß wird die NEW-Routine angesprungen, und von da aus geht es in den BASIC-Interpreter, der nun auf Ihre Eingaben wartet.

Unterbrechungen . . .

Die beiden anderen erwähnten Hardware-Unterbrechungen heißen „nicht maskierbarer Interrupt" (NMI) und Interrupt-Anfrage („Interrupt-Request", IRQ). Sie laufen ganz ähnlich ab wie ein Reset, dienen aber anderen Zwecken. Mit ihnen kann ein laufendes Programm unterbrochen werden, damit der Prozessor zwischendurch etwas anderes tut und dann zum unterbrochenen Programm zurückkehrt.

Beim C64 wird zum Beispiel regelmäßig ein IRQ ausgelöst, und zwar sechzigmal pro Sekunde. Das heißt, jedes Programm wird alle sechzigstel Sekunde unterbrochen. Der Prozessor führt dadurch regelmäßig ein Programm aus, das überprüft, ob eine Taste gedrückt ist, den Cursor blinken läßt, die Zeit (TI$) hochzählt und beim Drücken der STOP-Taste ein BASIC-Programm abbricht. Diese Routine läuft quasi „im Hintergrund", der Anwender merkt überhaupt nicht, daß sein Programm laufend unterbrochen wird.

Die Rückkehr zum unterbrochenen Programm wird dadurch bewerkstelligt, daß die CPU bei einem Interrupt ähnlich reagiert wie bei einem Unterprogrammaufruf. Die

Adresse des nächsten Befehles wird nämlich auf dem Stack abgelegt.

Zusätzlich speichert der Prozessor auch den Inhalt des Statusregisters auf dem Stack ab. Denn im Gegensatz zu einem JSR-Befehl weiß man ja als Programmierer nicht, wann während des Programmlaufes ein Interrupt ausgelöst wird. Kommt er zufällig zwischen einem CMP- und einem BEQ-Befehl und verändert die Interrupt-Routine die Zero-Flagge, könnte das fatale Folgen haben.

Zur Rückkehr in das unterbrochene Programm dient der Befehl

RTI (ReTurn from Interrupt),

der die Rückkehradresse und den Inhalt des Statusregisters wieder vom Stack holt.

. . . und wie man sich dagegen wehrt

Der Unterschied zwischen einem NMI und einem IRQ besteht darin, daß ein NMI in jedem Falle zu einer Programmunterbrechung führt, während man den IRQ unterbinden kann. Dazu dient die I-Flagge des Statusregisters. Sie kann mit dem Befehl

SEI (SEt Interrupt disable)

gesetzt werden. Dadurch wird die Annahme des Interrupt-Requests verhindert. Um den Interrupt wieder zuzulassen, muß das I-Flag gelöscht werden. Dazu dient der Befehl

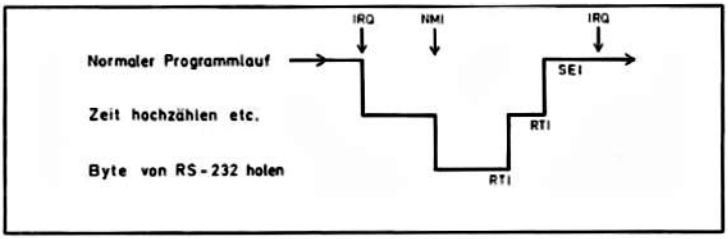
CLI (CLear Interrupt disable).

Ist die I-Flagge gelöscht, so ist die Reaktion auf einen IRQ die gleiche wie auf einen NMI. Im einzelnen passiert dabei folgendes: Zuerst wird der gerade in der Abarbeitung befindliche Befehl vollständig ausgeführt. Dann rettet der Prozessor die Rückkehradresse und das Statusregister auf den Stack und setzt die I-Flagge. Damit ist zunächst die Annahme eines weiteren IRQ gesperrt. Abschließend wird die Einsprungadresse der Interrupt-Routine aus dem ROM in den Programmzähler geladen und damit – genau wie bei einem Reset – die Interrupt-Behandlungs-Routine angesprungen.

Die Adresse, an der die IRQ-Routine beginnt, steht in den Speicherzellen $FFFE und $FFFF, die für den NMI bei $FFFA und $FFFB.

Beim C64 wird ein NMI durch den gleichzeitigen Druck auf die Tasten RUN/STOP und RESTORE ausgelöst. Er dient als „weicher Reset". Außerdem kann über eine am User-Port angeschlossene RS-232-Schnittstelle ein NMI ausgelöst werden.

Wie schon erwähnt, wird ein IRQ regelmäßig durch einen Timer in einem der beiden Schnittstellenbausteine (CIAs) ausgelöst. Aber es gibt im 64er noch andere Interrupt-Quellen. So können die CIAs so programmiert werden, daß sie beim Empfangen eines Bytes am seriellen Port oder beim Anlegen eines bestimmten Spannungspegels an einer speziellen Leitung einen IRQ auslösen. Das wird zum Beispiel beim Laden von der Datassette ausgenutzt. Auch der VIC (der für die Bilderzeugung zuständige Chip) kann IRQs auslösen, beispielsweise, wenn zwei Sprites sich berühren oder wenn der Elektronenstrahl auf der Bildröhre eine bestimmte Position erreicht.



Ein NMI kann die IRQ-Routine unterbrechen, aber nicht umgekehrt. Er hat die höhere Priorität.

Umleitung

Nun wäre es unsinnig, auf all diese Ereignisse in derselben Art und Weise zu reagieren. Darum enthalten die NMI- und die IRQ-Routine ziemlich am Anfang einen indirekten Sprungbefehl. Das ist ein JMP-Befehl, dessen Argument nicht direkt die Adresse ist, zu der gesprungen wird, sondern eine Adresse, an der diese Adresse steht.

Die IRQ-Routine enthält zum Beispiel den Befehl

JMP ($0314).

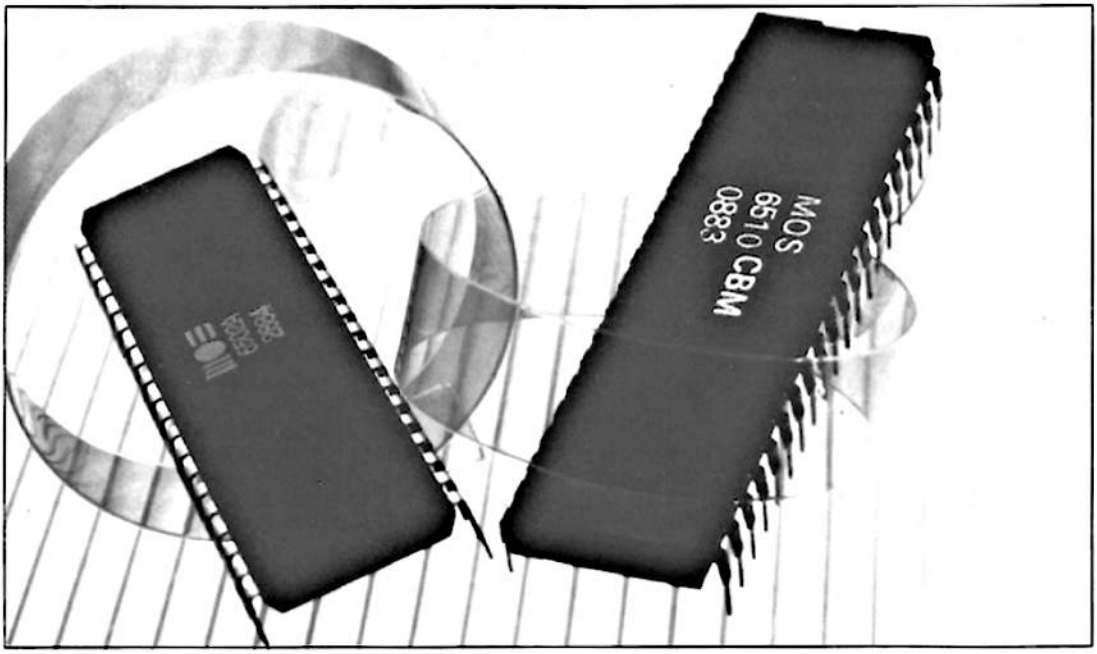
Wenn die CPU auf diesen Befehl trifft, springt sie nicht zu der Adresse $314, sondern holt sich die Sprungadresse aus den Speicherzellen $314 und $315 (788/789). Man sagt, diese Adressen enthalten den IRQ-Vektor. Normalerweise steht dort die Adresse $EA31, der Beginn der IRQ-Routine im ROM.

Will man nun Interrupts für eigene Anwendungen einsetzen, so muß man diesen Vektor auf seine eigene Interrupt-Behandlungsroutine „verbiegen". Wie das geht und was dabei zu beachten ist, können Sie einem ausführlich kommentierten Assembler-Quelltext entnehmen, den Sie aus dem Programm heraus abspeichern und mit dem INPUT-ASS wieder laden können. Es handelt sich dabei um ein Programm, das den VIC so programmiert, daß er beim Erreichen einer bestimmten Rasterzeile einen Interrupt auslöst. Die Interrupt-Routine schaltet dann die Hintergrund-Farbe um.

Auf der Speicherseite drei, genauer: auf den Adressen von $300 bis $33E, stehen übrigens noch andere Sprungvektoren. Mit Ihnen ist es möglich, alle wichtigen Routinen des C64-Betriebssystems auf eigene Programme umzuleiten. Damit kann man BASIC-Erweiterungen einbinden oder die Drucker-Ausgabe auf eine Centronics-Schnittstelle am Userport umleiten. Einige dieser Vektoren sollen in der nächsten Folge der Assembler-Schule vorgestellt werden.

Dann werden wir uns auch mit der Einbindung von Assembler-Routinen in BASIC-Programme beschäftigen und einen Blick auf die Programmierung von C64-Spezialitäten wie Joystick-Abfrage und Grafik werfen.

Hajo Schulz



Teamwork

INPUT 64-AssemblerSchule, Teil 6

Die einfachste Art, Maschinensprache-Programme zu starten, kennt sicherlich jeder von Ihnen: Der BASIC-Befehl SYS lädt die angegebene Adresse in den Programmzähler der CPU und führt so eine an dieser Stelle stehende Assembler-Routine aus. Damit kann man komplette Maschinenprogramme wie Spiele oder die Assembler-Schule starten. Will man aber kleine in Assembler geschriebene Hilfsroutinen einsetzen, ist in aller Regel eine Übergabe von Parametern an die Routine erforderlich. Auch die Ergebnisse dieser Routine sollen an das BASIC-Programm zurückgegeben werden.

Ging es bislang in diesem Kurs darum, die Befehle der 6502-CPU kennenzulernen und daraus Programme zusammenzusetzen, so werden wir uns in dieser abschließenden Folge darum kümmern, wie diese Programme benutzt werden können. Dabei geht es um Speicherbereiche, in denen Assemblerprogramme laufen können, aber auch um die Möglichkeiten der Parameterübergabe an Maschinensprache-Routinen und deren Zusammenarbeit mit BASIC-Programmen.

This is SYS

Dazu gibt es verschiedene Möglichkeiten. Sehen wir uns zunächst etwas genauer an, was der BASIC-Interpreter tut, wenn er einen SYS-Befehl abarbeitet. Bevor nämlich der Sprung in das Maschinensprache-Programm erfolgt, werden erst einmal die CPU-Register in einen definierten Zustand versetzt. Dazu dienen die Speicherstellen 780 bis 783 ($30C bis $30F). Mit deren Inhalt werden gemäß der Tabelle der Akku, die Index- und das Statusregister versorgt. Besonders die Adresse für das Status-Register ist hierbei mit Vorsicht zu genießen; eine gesetzte I- oder D-Flagge kann zu Problemen führen.

Für den eigentlichen Sprung in das Maschinenprogramm benutzt der BASIC-Interpreter einen JSR-Befehl. Dadurch kann man eine Maschinensprache-Routine mit dem RTS-Kommando abschließen und landet so wieder im BASIC-Programm, genau hinter

dem SYS-Befehl. Vorher werden aber die Register wieder in die erwähnten Speicherstellen gerettet.

Man kann also mit dem BASIC-Befehl POKE vor dem Einsatz eines SYS-Kommandos die CPU-Register setzen. Hinterher können Sie sich mit PEEK ansehen, welchen Inhalt sie am Schluß der Routine hatten, und so Ergebnisse übergeben bekommen.

Braucht eine Routine mehr Parameter, als in den drei Registern Platz haben (das Statusregister ist ja nicht uneingeschränkt verwendbar), muß man noch andere Speicherzellen dazunehmen. Auf diese kann das Maschinenprogramm dann explizit zugreifen.

Übergabe programmiert . . .

Diese Art der Kommunikation zwischen Haupt- und Unterprogrammen ist zwar sehr einfach zu realisieren, aber relativ unkomfortabel zu bedienen. Das Unterprogramm in Maschinensprache ist schnell geschrieben. Aber durch die vielen POKE- und PEEK-Befehle wird das BASIC-Hauptprogramm sehr unübersichtlich und auch langsam.

780 / $30C	Akkumulator
781 / $30D	X-Register
782 / $30E	Y-Register
783 / $30F	Prozessor-Status

Vier Speicherstellen dienen als Interface für den SYS-Befehl.

Ein Beispiel soll dies verdeutlichen. Im Betriebssystem des C64 gibt es an der Adresse $FFF0 (65520) eine Routine, die den Cursor an eine bestimmte Bildschirmposition setzt. Mit ihr kann man etwas an einer definierten Stelle ausgeben. Diese Routine kann auch die derzeitige Cursor-Position als Ergebnis ermitteln – soll der Cursor gesetzt werden, so muß beim Eintritt in die Routine die Carry-Flagge gesetzt sein. Die Routine erwartet die Zeile, in die der Cursor gesetzt werden soll, im X- und die Spalte im Y-Register. Nach dem bisher Gesagten kann man sie also mit folgender BASIC-Befehlsfolge aktivieren:

```
POKE 781,Z : REM Zeile in X
POKE 782,S : REM Spalte in Y
POKE 783,PEEK(783) AND 254 : REM Carry clear

SYS 65520 : REM Cursor setzen
PRINT "Text" : REM an Cursor-Position ausgeben
```

Selbst wenn man diese Befehlsfolge in eine BASIC-Zeile schreibt und die REMs wegläßt, ist sie doch relativ unübersichtlich.

Wünschenswert wäre etwa ein Aufruf der Art

```
SYS SC,Z,S : REM Cursor setzen
PRINT "Text" : REM an Cursor-Position ausgeben
```

wobei in SC die Startadresse einer Set-Cursor-Routine gespeichert ist und Z und S die Zeile beziehungsweise die Spalte enthalten. Diese Routine ist in Assembler einfach zu formulieren, da das Betriebssystem des C64 verschiedene Routinen zur Verfügung stellt, mit denen man Argumente direkt aus dem BASIC-Programmtext lesen kann. So gibt es ab Adresse $AEFD eine Routine, die überprüft, ob das nächste Zeichen ein Komma ist. Anderenfalls wird die Meldung 'SYNTAX ERROR' ausgegeben.

Die Routine, die an der Adresse $B7EB beginnt, ist für unseren Zweck wie geschaffen: Sie liest zwei durch Komma getrennte Zahlen aus dem Programmtext. Die erste darf dabei ein Zwei-Byte-Wert sein (wir brauchen nur das Low-Byte) und wird als Pointer in den Adressen $14 und $15 abgelegt. Die zweite muß ein Ein-Byte-Wert sein und wird im X-Register geliefert. Bei beiden Zahlen darf es sich übrigens auch um beliebige numerische Ausdrücke handeln.

Die zugehörige Maschinensprache-Routine haben wir als Listing abgedruckt. Sie ist nur 14 Bytes lang, bewirkt aber eine wesentlich komfortablere Erstellung von BASIC-Programmen, die den Cursor direkt adressieren sollen. Zudem arbeitet ein BASIC-Programm mit dieser Routine wesentlich schneller, als wenn man drei POKE- und einen SYS-Befehl einsetzt.

. . . und automatisch

Ein anderer Befehl zum Starten einer Maschinensprache-Routine, der leider sehr selten verwendet wird, ist die BASIC-Funktion USR( ). Bei ihr wird die Startadresse des Assembler-Programms nicht explizit angegeben. Vielmehr gibt es an den Adressen $311 und $312 (785, 786) einen sogenannten USR-Vektor. Bei jedem Auftreten des USR-Befehles springt der Interpreter zu der in diesem Pointer angegebenen Adresse. Das Argument, das diese Funktion in Klammern übergeben bekommt, wird vorher ausgewertet und in einem speziellen Fließkommaformat in dafür vorgesehenen Adressen der Zero-Page abgelegt.

Diese Adressen sind für den BASIC-Interpreter reserviert und bilden den sogenannten Fließkomma-Akku (FAC). Er befindet sich an den Adressen $61 bis $66 und wird vom Interpreter für alle arithmetischen Funktionen benötigt, ähnlich dem Prozessor-Akkumulator, den die CPU ja in Assemblerprogrammen für Rechenoperationen benutzt.

Auch eine durch den USR-Befehl aktivierte Assembler-Routine kann mit RTS abgeschlossen werden. Damit nimmt der Interpreter seine Arbeit wieder an der Aufrufstelle im BASIC-Programm auf. Der USR-Befehl wird jedoch vom Interpreter wie eine

```
01

c000            org $c000

c000 20fdae     jsr $aefd       Auf Komma pruefen
c003 20e6b7     jsr $b7eb       Zwei Zahlen holen
c006 8a         txa
c007 a8         tay             Spalte in Y
c008 a614       ldx $14         Zeile in X
c00a 18         clc
c00b 4cf0ff     jmp $fff0       Cursor setzen
```

Vierzehn Bytes reichen für eine komplette PRINT-AT-Routine.

Funktion – ähnlich wie SIN( ) oder EXP( ) – behandelt. Das heißt, er kann nicht wie der SYS-Befehl allein stehen, sondern liefert ein Ergebnis, das vom Programm weiterverarbeitet werden muß. Erlaubt sind zum Beispiel Konstruktionen wie

PRINT USR(x/10) oder
DU-USR(13)

Das Ergebnis der USR-Funktion wird vom Interpreter wieder im FAC erwartet. Falls Sie das Ergebnis nicht benötigen, können Sie die USR( )-Funktion wie im zweiten Beispiel anwenden. Die Variable DU dient dann lediglich als Dummy.

Joystick-Akrobatik

Sie können aus Ihrem Programm heraus einen Quelltext für den INPUT-ASS (Ausgabe 6/86) abspeichern, der die Verwendung der USR-Funktion an einem Beispiel demonstriert. Der USR-Vektor wird dabei auf eine Routine gelenkt, die den Joystick abfragt. USR(1) liefert als Ergebnis 1, wenn der Stick nach rechts gedrückt ist, –1 bei einer Bewegung nach links und 0 wenn er sich in Mittelstellung befindet. Mit USR(2) kann die y-Richtung abgefragt werden, 1 bedeutet runter und –1 hoch. USR(3) ergibt 1, wenn der Feuerknopf gedrückt ist, sonst 0.

Das Programm beginnt mit einigen Bytes, die eine BASIC-Zeile darstellen. Jede BASIC-Zeile besteht aus einem Zeiger auf die nächste Zeile, der Zeilennummer in Low-High-Byte-Format und dem eigentlichen Text. Abgeschlossen wird sie durch ein Null-Byte. Das Programm liegt in sogenanntem Interpreter-Code vor, bei dem die BASIC-Schlüsselworte nicht als Klartext, sondern als Tokens (ein 17 Byte pro Wort) erscheinen. Der SYS-Befehl wird zum Beispiel als $9E verschlüsselt.

BASIC mal woanders

Durch die S-Anweisung wird sichergestellt, daß das eigentliche Programm auch wirklich an der in der BASIC-Zeile angegebenen Adresse beginnt. Dort wird zunächst der USR-Vektor „verbogen". Die folgenden Befehle dienen dazu, den BASIC-Anfang hochzusetzen. Mit dieser Formulierung ist folgendes gemeint: Normalerweise beginnt ein BASIC-Programm an der Adresse $801 (2049) im Speicher. Diese Adresse ist aber nicht fest, sondern wird durch den Inhalt der Speicherzellen $2B und $2C (43, 44) bestimmt. Sie enthalten einen Pointer auf das erste Byte des BASIC-Speichers.

Unser Beispielprogramm läßt sie auf das Label NEW, das letzte in diesem Programm, zeigen. Dadurch ist der Platz, den unsere USR-Routine belegt, vor Zugriffen durch den BASIC-Interpreter geschützt. Für ihn beginnt der Speicher praktisch erst an der Adresse NEW. Die beiden folgenden Betriebssystem-Aufrufe dienen dazu, auch andere interpreterspezifische Speicherstellen auf die neue Konfiguration einzustellen und ein eventuell an der Adresse NEW beginnendes BASIC-Programm zu starten.

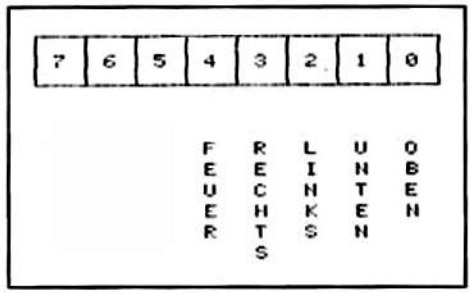
So wie der Assembler das Programm auf Diskette schreibt, ist natürlich an dieser Stelle noch kein BASIC-Programm vorhanden. Wenn Sie aber das erzeugte Programm mit RUN starten, landen Sie wieder im BASIC-Direktmodus und können nun ein Programm eingeben oder laden, das die neuartige USR-Routine benutzt. Speichern Sie dieses Programm auf einen Datenträger, nachdem Sie mit der Befehlsfolge

POKE 43,1 : POKE 44,8

den BASIC-Anfang wieder auf den ursprünglichen Wert zurückgesetzt haben, dann wird beim erneuten Laden und Starten mit RUN zuerst wieder der USR-Vektor auf die Joystick-Routine gerichtet, und dann beginnt das BASIC-Programm zu arbeiten.

Ein Bit pro Richtung

Sehen wir uns nun die eigentliche Joystick-Abfrage an: Sie beginnt mit dem Aufruf einer Routine im BASIC-Interpreter, die die im FAC befindliche Fließkommazahl in einen Integer-Wert umrechnet. Sie liefert das High-Byte im Akku und das Low-Byte im Y-Register. Da unsere USR-Routine nur für die Argumente 1, 2 und 3 gedacht ist, muß das High-Byte gleich Null sein. Anderenfalls soll sie die Fehlermeldung „ILLEGAL QUANTITY ERROR" erzeugen.



Das Datenregister A der CIAs ist zuständig für die Joystick-Anschlüsse.

Der SEI-Befehl, der nun folgt, verhindert für einige Zeit die Aktivierung der Tastaturabfrage in der Interrupt-Routine. Das ist notwendig, weil die Tastatur an den gleichen Ports der Peripherie-Chips (CIA, Complex Interface Adapter) hängt wie die Joystick-Stecker. Mit dem Zugriff auf das Datenrichtungsregister wird die CIA so programmiert, daß die für den Joystick zuständigen Bits (siehe Bild) als Eingang zur Verfügung stehen. Das Bitmuster landet im Akku. Anschließend wird die CIA in ihren Originalzustand versetzt und die Tastaturabfrage wieder zugelassen.

Im Y-Register steht immer noch die Zahl, mit der die USR-Routine aufgerufen wurde. Je nach ihrem Wert verzweigt das Programm nun zu verschiedenen Stellen, um die unterschiedlichen Richtungen abzufragen. Um diese Abfrage zu verstehen, müssen Sie noch wissen, daß das für den jeweiligen Joystick-Kontakt zuständige Bit Null wird, wenn der Kontakt geschlossen ist.

Das BASIC-Programm soll ja im FAC eine der Zahlen – 1, 0 oder 1 zurückbekommen. Um eine Zahl in den FAC zu bekommen, benutzen wir die Routine TOFAC ($BC44), die in der Zero-Page-Speicherzelle $62 das High-Byte und in $63 das Low-Byte einer Integerzahl erwartet. Ferner muß das X-Re-

gister $90 enthalten. Diese Routine endet mit einem RTS. Mit diesem Befehl landet der Prozessor wieder im Interpreter. Den Trick, direkt aufeinanderfolgende JSR-und RTS-Befehle durch ein JMP zu ersetzen, kennen Sie ja schon.

Das zweite Source-File, das Sie sich aus dem Programm abspeichern können, enthält eine echte BASIC-Erweiterung. Mit diesem Programm wird es möglich, in BASIC-Programmen direkt Hexadezimalzahlen zu verwenden.

Hexerei

Diese Art der BASIC-Erweiterungen wird dadurch ermöglicht, daß der BASIC-Interpreter zwei Vektoren auf der Seite 3 benutzt. Immer wenn er im Programmtext ein Kommado (PRINT, POKE, RETURN etc.) erwartet, benutzt er den Befehl JMP ($308).

Normalerweise zeigt der Vektor $308/$309 auf die Adresse $A7E4. Dort beginnt eine Routine, mit deren Hilfe der Interpreter für jeden Befehl eine entsprechende Adresse findet, zu der er springt, um eben diesen Befehl zu behandeln. Ein zweiter wichtiger Vektor liegt auf den Adressen $30A/$30B. Dieser wird vom Interpreter immer dann verwendet, wenn er erkennt, daß ein auszuwertender Ausdruck folgt. Das kann eine Variable, ein Rechenausdruck oder ein String sein.

Genau diesen Vektor, der normalerweise auf die Adresse $AE86 zeigt, verwenden wir in unserem Beispiel. Er wird auf die Routine FUNCTION gerichtet. Die für das Verbiegen zuständigen Befehle sehen etwas umfangreich aus, sind aber aus folgenden Gründen notwendig:

Der Interpreter springt wie gesagt bei jedem Ausdruck über diesen Vektor. Wenn es sich dabei nun nicht um eine Hexadezimalzahl handelt, muß er ihn wie gewohnt behandeln, der ursprüngliche Inhalt des Vektors muß also gerettet werden. Dabei muß es sich nicht notwendigerweise um die Adresse $AE86 handeln, es kann ja vorher schon eine andere BASIC-Erweiterung geladen gewesen sein.

Die eigentliche Routine, auf die der Vektor für Ausdruckauswertung nun zeigt, beginnt mit einem Aufruf der Routine GETCHR. Mit dieser Routine liest der Interpreter Zeichen für Zeichen aus dem BASIC-Programmtext. Sie ist im Programm auf Ihrem Datenträger ausführlich erklärt.

Handelt es sich bei dem gelesenen Zeichen nicht um ein '$', so wird der Eingangszustand wiederhergestellt und zu der bei der Initialisierung geretteten Adresse gesprungen. Anderenfalls wird die Routine HEXIN aktiviert, die Sie anhand der Kommentare verstehen sollten. Die Umrechnung von Ziffern und Buchstaben in Hex-Werte ist die Umkehrung dessen, was Sie aus früheren Folgen unter dem Stichwort PBYT kennen.

Die Routine bricht ab, wenn ein Zeichen erkannt wird, das keine Hex-Ziffer darstellt. Zum Schluß wird das Ergebnis wieder mittels der Routine TOFAC in den Fließkomma-Akku übertragen. Daß die Adresse für TOFAC in diesem Programm eine andere als im ersten Beispiel ist, liegt daran, daß im ersten Beispiel auch ein negatives Ergebnis auftreten konnte. Hexadezimalzahlen sollen aber normalerweise nicht vorzeichenbehaftet sein. Wenn Sie wollen, daß diese Routine bei Zahlen größer als $7FFF ein negatives Ergebnis liefert, können Sie hier auch die Adresse aus dem ersten Beispiel einsetzen.

No Future?

An dieser Stelle endet unsere Assembler-Schule. Wir haben versucht, Ihnen die Grundlagen der Maschinensprache-Programmierung nahezubringen. Wenn Sie alle Folgen durchgearbeitet haben, sollten Sie in der Lage sein, zumindest Assemblerprogramme zu lesen und selbst kleinere Routinen in Maschinensprache zu entwickeln.

Natürlich kann eine sechsteilige Serie nicht alle Aspekte dieser Art und Weise, Programme zu entwickeln, behandeln. Aber bei Assemblerprogrammierung ist es wie bei allen Dingen, die man neu lernt: Übung macht den Meister und Probieren geht über Studieren.

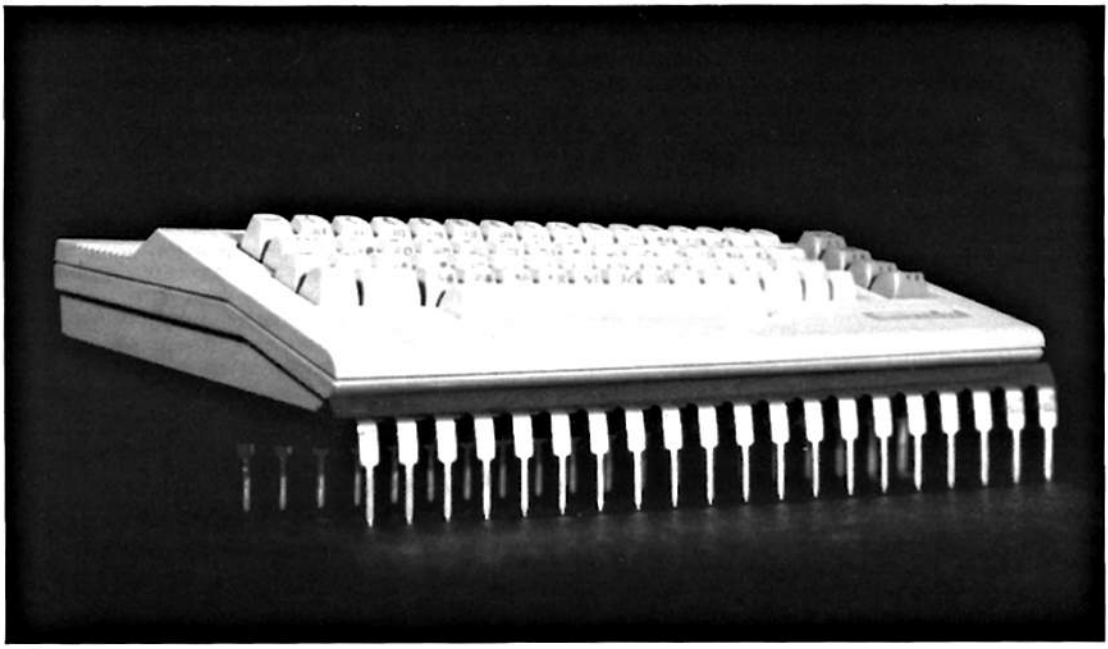
Im übrigen werden Sie in INPUT 64 auch weiterhin Tips zur Assemblerprogrammierung und kommentierte Quelltexte finden.

Hajo Schulz

Literatur

Christian Persson: 6502/65C02-Maschinensprache, Verlag Heinz Heise GmbH, Hannover 1983

Rodnay Zaks: Programmierung des 6502, Sybex-Verlag GmbH, Düsseldorf 1981



Ganz schnell wird's maschinell

Makro-Assembler/Editor für C64 und C128

Als INPUT-ASS ist dieser Assembler treuen Lesern unseres Magazins bereits seit Juni 1986 bekannt. Wir haben dieses für Maschinensprache-Programmierer unentbehrliche Werkzeug für den 128er angepaßt — mit 40/80-Zeichen-Modus wahlweise und voller Nutzung des größeren Speichers. Und damit die „Nur-64er" nicht zu kurz kommen, finden Sie auf Diskette auch einen INPUT-ASS-64, in den fast zweijährige Redaktions- und Lesererfahrung sowie „Debugging" einflossen.

INPUT-ASS-64 wird von der eigenen Diskette (Sie müssen ihn natürlich aus dem Magazin heraus zunächst abspeichern) wie ein BASIC-Programm mit LOAD"INPUT-ASS-64",8,0 geladen und mit RUN gestartet; die 128er-Version entsprechend mit DLOAD"INPUT-ASS-128" und RUN. Die C128-Version erkennt selbständig, ob der 40- oder 80-Zeichen-Bildschirm bedient werden will. Zwischen den beiden Bildschirmen kann ohne Textverlust hin- und hergeschaltet werden, aber nur außerhalb des Assemblers (also: CTRL-K/Q, ESC und 'X', RUN).

Die Bezeichnung Assembler/Editor deutet schon darauf hin, daß es sich genaugenommen um zwei Programme handelt: den Texteditor zum Erstellen, Laden, Speichern und Ändern der Programmtexte (der natürlich auch für beliebige andere Texte benutzt werden kann) und den Assembler, der den Source-Code in Maschinenbefehle übersetzt (siehe auch „Wie das geht . . ."). Vorweg noch eine Bemerkung zur Schreibweise: alle Mnemonics, Pseudo-OPs und Fehlermeldungen (LDA, PHC, AM und so weiter) sind in diesem Text der Übersichtlichkeit halber in VERSALIEN geschrieben. Der Assembler erwartet aber grundsätzlich Kleinschrift.

Der Editor — beinahe eine Textverarbeitung

Der Editor dürfte denjenigen, die schon mit WordStar oder Turbo-Pascal gearbeitet haben, „irgendwie bekannt" vorkommen. Die Befehle sind tatsächlich weitgehend WordStar-kompatibel. Am einfachsten und

Was das ist, wie das geht und wozu das alles gut ist

INPUT-ASS ist ein 2-Pass-Makro-Assembler mit integriertem Editor und optionaler Speicher- oder Peripherie-Orientierung, der sowohl symbolische als auch berechnete Ausdrücke verarbeitet.

Klingt gut, nicht? Wenn Sie gerade Ihr hundertunderstes Assemblerprogramm schreiben, sowieso wenig Zeit haben und Begriffe wie Vorwärts-Referenz und Phase-Error im Schlaf erklären können, lesen Sie bitte nur die Kurzübersicht. (Wahrscheinlich lesen Sie dann nicht mal die, weil Sie zu den Leuten gehören, die Anleitungen erst nach dem fünften System-Crash zwischen den Comics hervorholen, oder?) Für alle anderen also:

Das Prinzip

INPUT-ASS verwandelt die für den Menschen verständlichen Befehle der Assemblersprache in für den Prozessor verständlichen 6502-Maschinencode. Aus dem Befehl JMP $FCE2 (Sprung zur Adresse 64738) wird die Byte-Folge $4C,$E2,$FC. Klartextbefehle wie JMP, LDA und so weiter werden übrigens „Mnemonics"[1] genannt. Nun sind – um gleich auf das Stichwort „Symbolverarbeitung" zu kommen – Adressen, Bytes und Bits genau das, was ein Computer gern verarbeitet. Da der Mensch aber den Dingen lieber einen Namen gibt, verarbeitet jeder vernünftige Assembler Symbole. Ein Befehl wie JMP RESET klingt doch wesentlich deutlicher als das tumbe JMP $FCE2.

Von „berechneten Ausdrücken" war oben die Rede. Das heißt einfach, daß der Assembler für Sie das tut, was er sowieso besser kann als Sie, nämlich mehr oder weniger komplizierte Rechenoperation durchführen. Wenn beispielsweise in die zehnte Reihe und fünfte Spalte des Bildschirms ein Zeichen geschrieben werden soll, kann man als Adresse angeben: 10*40+5+VIDEO. Natürlich muß dem Assembler vorher „mitgeteilt" werden, welchen Wert VIDEO hat, etwa durch die Zuweisung VIDEO=$400.

Wie erledigt ein Assembler nun seine eigentliche Arbeit, die Umwandlung von Programmtext in Maschinencode? Stellen Sie sich einmal vor, Sie seien der Assembler. Ihnen wird Zeile für Zeile Source-Text (nicht neudeutsch: Quelltext, besseres Wort für Programmtext) vorgehalten, den Sie in den entsprechenden Maschinencode verwandeln sollen. Das ist in solchen Fällen nicht besonders schwierig, wo es nur um die Zuordnung von Mnemonics wie LDA zum Code $A9 geht. Das kann man in einer „Kartei" (sprich Tabelle) zuordnen, und Zahlen werden ohnehin nur in ihre entsprechende Bit-Kombination umgewandelt. Labels (Sprungmarken, Einsprungspunkte für Unterprogramme und so weiter) sind dann kein Problem, wenn Sie zum Zeitpunkt des Aufrufs bekannt sind. Beispiel:

```
ORG $C000
:WAIT DEX
BNE WAIT
```

Die Anfangsadresse ist durch die ORG-Anweisung bekannt. Deswegen können Sie das Label WAIT eindeutig der Adresse $C000 zuordnen. Interessant wird es bei dem indirekten Sprung BNE WAIT. Sie – immer noch in der Rolle des Assemblers – müssen jetzt feststellen, an welcher Adresse WAIT liegt. Dies ist, werden Sie sagen, kein Problem, denn dem Symbol WAIT wurde bereits vor (!) dem Befehl BNE WAIT die Adresse $C000 zugeordnet. Richtig. Sie mußten sich diese Zuordnung nur merken. Darum jetzt der schwierige Fall:

```
ORG $C000
:BEGIN LDX #100
JSR WAIT
...
...
:WAIT DEX
BNE WAIT
RTS
```

Beim Interpretieren des Programmtextes stoßen Sie auf den Befehl JSR WAIT – und haben nicht den Schimmer einer Ahnung, an welcher Adresse sich WAIT befinden könnte, denn es taucht ja wesentlich später im Text auf. (Und heißt deswegen „Vorwärtsreferenz"!) Was tun? Die gängige Möglichkeit ist die: sich erstens die Stellemerken, an der das noch unbekannte Label auftauchte. Zweitens, so zu tun, als ob nichts gewesen wäre, und den Programmzähler um die richtige Länge erhöhen. Denn es ist bekannt, daß ein JSR-Befehl immer drei Bytes lang ist. Beim weiteren Durchsuchen des Assemblertextes stoßen Sie dann irgendwann auf das Label WAIT und merken sich natürlich dessen Adresse. (Dieses Merken, also die Zuordnung von Labels beziehungsweise Symbolen, nennt man übrigens fachgerecht „Generieren einer Symboltabelle").

schnellsten können Sie die Befehle kennenlernen, indem Sie alle ausprobieren.

Bildschirmaufteilung: Die oberste Zeile des Bildschirms ist die Statuszeile. Ein inverses Zeichen am linken Rand zeigt an, daß ein aus zwei Zeichen bestehender Befehl eingegeben wird. Die Hex-Zahl daneben gibt die Größe des noch freien Speichers an. Der Rest der Statuszeile ist vorgesehen für Eingaben, Statusmeldungen der Floppy und Fehlermeldungen des Assemblers. Der Text wird in den restlichen 24 Zeilen des Bildschirms dargestellt. Die Zeilen sind 80 Zeichen lang, durch horizontales Scrolling sind jeweils 40 von 80 Zeichen der Zeile sichtbar. Der Cursor wird invers angezeigt.

Texteingabe: Die maximale Zeilenlänge beträgt 79 Zeichen, danach sind keine Eingaben mehr möglich. Die Zeilen werden durch RETURN abgeschlossen. Bei der Eingabe gibt es zwei Möglichkeiten (durch ein Flag in der linken oberen Ecke angezeigt): Im Insert-Modus (I) werden die eingegebenen Zeichen automatisch eingefügt, im Overwrite-Modus (O) die unter dem Cursor liegenden Zeichen überschrieben.

Alle Befehle, die Peripherie-Operationen betreffen (Laden, Speichern, Drucken), sind blockorientiert. Das heißt, daß immer der

„Aber", werden Sie einwenden, „wann trage ich denn die echte Adresse von WAIT nach dem JSR-Befehl ein?" Sie ahnen es sicherlich schon: Sie müssen ein zweites Mal durch. Wenn Sie dann auf den Befehl JSR WAIT stoßen, sehen Sie in der Symboltabelle nach, ob es schon eine Zuordnung zu WAIT gibt. Wenn ja, wird die entsprechende Adresse eingesetzt. Wenn nicht, hauen Sie – nach wie vor in der Rolle des Assemblers – dem schlampigen Programmierer eine Fehlermeldung um die Ohren.

Das erklärt wohl auch den Begriff „2-Pass-Assembler": damit ist nichts anderes als das eben beschriebene zweimalige Bearbeiten des Programmtextes gemeint, so daß erst im zweiten Durchgang (Pass) Maschinencode erzeugt wird.

Die oben erwähnte „optionale Speicher- oder Peripherie-Orientierung" hat zu tun mit dem eingebauten Editor. Der Editor ist eigentlich ein eigenes Programm, nämlich eine zeilenorientierte Textverarbeitung. Ihr aus den oben erwähnten „Mnemonics", Zuweisungen und (hoffentlich!) Kommentaren bestehender Source-Code (wörtlich übersetzt: Quell-Code, also der vom Assembler zu verarbeitende Text) wird mit diesem Editor erstellt. Näheres zum Editor siehe unten, hier soll es nur um die Beziehung zwischen Editor und Assembler gehen.

Makros und Include-Files

Im Prinzip erwartet der Assembler den Text im Editor. Aber eben nur im Prinzip. Es gibt nämlich zwei Spezialfälle:

Erstens: Makros. Makros sind, theoretisch ausgedrückt, „Text-Ersatz-Anweisungen".

Ihr praktischer Nutzen besteht darin, ständig wiederkehrende Befehlsfolgen nicht immer neu eintippen zu müssen. Dazu wird einmal ein Makro definiert; diese Definition besteht aus

- dem Namen des Makros
- einem Befehl, der angibt: „Jetzt wird ein Makro definiert" (beim INPUT-ASS 'm' oder 'mf')
- gegebenenfalls der Anzahl der Parameter, die diesem Makro übergeben werden sollen
- der diesem Makro zuzuordnenden Befehlsfolge
- und, natürlich, einer Endemarkierung.

Findet der Assembler nach dieser Makro-Definition im Text den Namen dieses Makros, so „tut er so", als ob der vollständi- schinencode erzeugt. Makros sind also keinesfalls ein Ersatz für Unterprogramme, die ja nur einmal im Code vorhanden sind und beliebig oft aufgerufen werden können.

Zweitens: Include-Files. Durch Include-Files können häufig benutzte Unterprogramme oder Systemdefinitionen bequem eingebaut werden. Außerdem ist dadurch die Länge von Source-Texten prinzipiell unbegrenzt. Include-Files sind Source-Texte, die während des Assemblierens von einem Datenträger Zeile für Zeile eingelesen werden. Der entsprechende Maschinencode wird im erzeugten Programm dort eingefügt, wo das Include-File aufgerufen wurde.

Damit hätten wir den einen Fall von „optionaler Speicher- oder Peripherie-Orientierung" abgehandelt, nämlich die Frage: Woher kriegt der Assembler seinen Text? Da drängt sich natürlich eine andere Frage geradezu auf: Wohin soll der Assembler mit dem erzeugten Code? Beim INPUT-ASS gibt es drei Möglichkeiten.

Erstens: Der Code wird direkt in den Speicher geschrieben. Das ist praktisch zum Austesten, geht aber nur, wenn der betreffende Speicherbereich nicht belegt ist. In den Assembler oder in den Textspeicher zu assemblieren, führt natürlich zu bösen Fehlern.

Zweitens: Der Code wird auf Diskette geschrieben. Geht im Prinzip immer.

Drittens: Es wird kein Code erzeugt. Dies kann zum Test auf syntaktische Fehler im Source-Code sinnvoll sein oder dann, wenn sich Drucker und Floppy nicht vertragen.

Das soll an Theorie erst einmal reichen. Alles Konkrete finden Sie in der Bedienungsanleitung und – wie immer – in der Praxis. Für alle, die Maschinensprache erst noch lernen wollen, sei hier noch einmal auf INPUT 64-Special 2/Maschinensprache hingewiesen, das unter anderem eine sechsteilige interaktive Assembler-schule enthält. JS

[1]Dieser Zungenbrecher hat etwas mit „mnemonisch" zu tun, was in etwa „gedächtnisunterstützend" bedeutet.

aktuell markierte, invers dargestellte Block gedruckt oder abgespeichert wird. Dies kann natürlich auch der gesamte Text sein. Diese Blockorientierung erlaubt ein beliebiges Zusammenbauen und Auseinandernehmen des Programmtextes. Haben Sie zum Beispiel eine Eingaberoutine geschrieben, die als Unterprogramm für weitere Anwendungen geeignet ist – den entsprechenden Teil als Block markieren und abspeichern. Dieser Programmteil kann dann in andere Source-Texte wieder eingefügt werden. Oder Sie wollen nur bestimmte Teile ausdrucken – als Block markieren und über CTRL-K/CTRL-W (siehe unten) auf Geräteadresse 4 schreiben. INPUT-ASS unterstützt so die Möglichkeit, sich eine Programmbibliothek zusammenzustellen – Sie müssen dann nur noch entsprechend überlegt programmieren . . .

Alles unter CTRLIe

Sämtliche Befehle werden über die Eingabe von CTRL und gleichzeitig irgendeines anderen Zeichens erreicht. CTRL meint dabei die CTRL-Taste, CTRL-V bedeutet beispielsweise, die CTRL-Taste gedrückt zu halten und gleichzeitig (!) ein kleines 'V' einzugeben. Mit Cursor-Zeile beziehungsweise Cursor-Position ist immer die Zeile/Position ge-

meint, in der sich der Cursor gerade befindet.
Bei allen Befehlen, die aus zwei Zeichen bestehen, kann das zweite Zeichen mit oder ohne CTRL eingegeben werden. So bedeutet CTRL-K/W: zuerst gleichzeitig die CTRL-Taste und 'K' drücken, anschließend 'W' oder CTRL und 'W'.

CTRL-V Insert-Taste geht auch; schaltet zwischen Insert- und Overwrite-Modus um.

CTRL-P erlaubt die Eingabe von Kontrollzeichen in den Text, die sonst als Befehle interpretiert würden. Kontrollzeichen werden invers dargestellt. Beispiel:

CTRL-P und anschließend CTRL-C gibt CTRL-C in den Text ein.

Cursor-Bewegungen

Die Tasten zur Bewegung des Cursors sind als Kreuz angeordnet (natürlich können auch weiterhin die Cursor-Tasten benutzt werden!):

```
    W     E     R
  A   S   D   F
    Z     X     C
```

CTRL-S	Cursor links
CTRL-D	Cursor rechts
CTRL-A	Wort links
CTRL-F	Wort rechts
CTRL-E	Cursor rauf
CTRL-X	Cursor runter
CTRL-W	Scroll up
CTRL-Z	Scroll down
CTRL-R	Seite rauf
CTRL-C	Seite runter
CTRL-I	Tabulator
CTRL-Q/S	zum Zeilenanfang
CTRL-Q/D	zum Zeilenende
CTRL-Q/R	zum Textanfang
CTRL-Q/C	zum Textende
CTRL-Q/B	zum Blockanfang
CTRL-Q/K	Blockende

Der Begriff „Block" wird unten noch einmal aufgenommen. Der Tabulator ist nur ein „Pseudo-Tabulator": er bewegt den Cursor in die Spalte des nächsten Wortanfangs der vorigen Zeile. Dies funktioniert natürlich nicht, wenn die Zeile über dem Cursor eine Leerzeile ist. Klingt komplizierter, als es ist – ausprobieren!

Einfügen und Löschen

DEL Zeichen links vom Cursor löschen. Falls der Cursor am linken Rand steht, wird die Cursor-Zeile mit der vorigen Zeile verknüpft.

CTRL-G Zeichen unter dem Cursor löschen.

CTRL-Q/Y Von der Cursor-Position bis zum Zeilenende löschen.

CTRL-Y Cursor-Zeile löschen. Vorsicht: Das Auto-Repeat kann hier verheerende Auswirkungen haben!

CTRL-Q/L macht alle Änderungen in der Cursor-Zeile rückgängig, sofern die Zeile noch nicht verlassen wurde. Vorsicht: CTRL-Y wird dadurch nicht rückgängig gemacht!

Block-Befehle

Ein Block wird invers angezeigt, die Cursor-Zeile aber immer normal. Es gibt immer nur einen aktuellen Block.

CTRL-K/A Gesamten Text als Block markieren. Empfehlenswert vor dem Abspeichern!

CTRL-K/B Cursor-Zeile wird als Blockanfang markiert.

CTRL-K/K Cursor-Zeile wird als Blockende markiert, das heißt als erste Zeile, die nicht zum Block gehört.

Falls die Markierungen zu einem Widerspruch führen (Blockende vor Blockanfang), wird der Blockanfang auf den Textanfang beziehungsweise das Blockende auf das Textende gelegt.

CTRL-K/C kopiert den markierten Block an die Cursor-Position, als Block ist danach die Kopie markiert; der zuvor als Block markierte Text bleibt unverändert.

CTRL-K/V verschiebt den Block an die Cursor-Position; der zuvor als Block markierte Text wird gelöscht. Sehr große Blöcke müssen in Portionen verschoben werden, da CTRL-K/V zuerst den Block kopiert und dann das Original löscht.

CTRL-K/Y löscht den Block **endgültig**.

Eingaben in der Statuszeile

Wenn die Eingabe eines File-Namens oder eines Suchwortes erwartet wird, erscheint eine Frage (zum Beispiel „LOAD:") und der Cursor. In der Statuszeile kann nur mit DEL editiert werden. Die Eingabe wird mit RETURN abgeschlossen und mit STOP abgebrochen.

File-Namen werden in einem speziellen Format eingegeben:

LOAD:82NAME

dabei ist 8 die Gerätenummer und 2 die Sekundäradresse. Diese Zahlenangaben sind hexadezimal einzugeben, Gerätenummer 8 und Sekundäradresse 15 ist also 8F. Der Name wird ohne Anführungsstriche eingegeben. Beispiele:

40: Printer (kein File-Name !)
10NAME: File „NAME" von Kassette lesen
11NAME: File „NAME" auf Kassette schreiben
82NAME: File „NAME" von Diskette lesen
83NAME: File „NAME" auf Diskette schreiben

Mit CTRL-D kann nach der Eingabe diese als Default (festgelegte Vorgabe) gespeichert werden. Dieser Default erscheint dann jedesmal, wenn diese Eingabe gemacht werden soll.

File-Befehle

CTRL-K/E Disk-Befehl senden. Beispiel:

DISK:8FS:TEXT

löscht das File „TEXT" auf der Diskette (8F für 8,15 !).

CTRL-K/F Directory einlesen. Die Ausgabe kann mit der Leertaste angehalten und mit der C=-Taste wieder fortgesetzt werden. Am Schluß muß eine Taste gedrückt werden, um wieder in den Editor zu kommen.

CTRL-K/R File einlesen. Beispiel:

LOAD:82ASM.S

liest das File „ASM.S" von der Diskette ein. Der Text wird an der Cursor-Position eingefügt und als Block markiert.

CTRL-K/S Adreßbereich abspeichern (nur C64!). Gefragt wird nach Anfangs- (FROM:) und Endadresse+1 (TO:) – vierstellige Hex-Zahlen gefordert – und dem File-Namen.

CTRL-K/W Block als File schreiben. Beispiel:

SAVE:82ASM.S

speichert den markierten Block unter dem

Namen „ASM.S" auf Diskette. Wenn es verdächtig schnell ging, haben Sie wahrscheinlich vergessen, den Block zu markieren, und die Warnung NO BLOCK übersehen . . .

CTRL-K/* schreibt den Editor mitsamt Text und Defaults auf die Diskette. Dies geht nur mit Diskette, da ein Daten-File geschrieben wird. Beim INPUT-ASS-128 wird durch diesen Befehl **nicht** der Source-Text mit abgespeichert, sondern nur der Assembler mit den Defaults. Mit diesem Befehl kann ohne Kopierprogramm eine angepaßte Version vom INPUT-ASS auf die Arbeitsdiskette geschrieben werden.

Nach den Disketten-Operationen wird in der Statuszeile der Disk-Status angezeigt.

File-Format

Der Editor speichert die Texte als SEQ-Files ohne jegliche Zusätze, das heißt, die Zeilen werden durch CR (Carriage Return, entspricht CHR$(13)) begrenzt. CHR$(0) ist verboten, da der Editor das als Endmarke verwendet. Zeilen sollten – CR inbegriffen – nicht länger als 80 Zeichen sein. Texte von Assemblern, die den BASIC-Editor verwenden (herzliches Beileid . .), können so lesbar gemacht werden:

```
LOAD „SOURCE",8
OPEN 1,8,2,„DEST,S,W"
CMD 1:LIST
PRINT #1:CLOSE 1
```

Ein vorheriges POKE 22,35 erzeugt übrigens ein Listing ohne Zeilennummern. Außerdem befinden sich sowohl auf der im Verlag erhältlichen Source-Diskette (siehe Anzeige in diesem Heft) als auch auf INPUT 64-Special 2 verschiedene Source-Code-Wandler.

Suchen und Ersetzen

Die Suchwörter beziehungsweise Ersatzwörter werden in der Statuszeile eingegeben.

CTRL-J Label-Definition suchen, zum Beispiel setzt

JUMP::LABEL

den Cursor auf die Label-Definition :LABEL. Dieser Befehl durchsucht den ganzen Text,also nicht nur vorwärts wie CTRL-Q/F oder CTRL-Q/A.

CTRL-Q/F Beliebige Zeichenfolge finden. Dabei wird ab Cursor-Position der Text bis zum Ende durchsucht. Der Cursor bleibt jeweils am Ende des gefundenen Wortes stehen. Sollen auch Zeichenfolgen vor der aktuellen Cursor-Position gefunden werden, empfiehlt sich ein vorheriges CTRL-Q/R.

CTRL-Q/A Suchen und Ersetzen. Beispiel:

```
FIND:ASC
CHNG:B
```

ersetzt 'ASC' durch 'B'. Bei jeder Fundstelle fragt der Computer:

REPLACE (Y/N/*)?

Die möglichen Antworten bedeuten:

'Y' = ersetzen
'N' = nicht ersetzen
'*' = automatisch ersetzen
STOP = Abbruch

Durch '*' werden also sämtliche weiteren Suchworte durch das Ersatzwort ersetzt, ohne nochmalige Abfrage.

CTRL-L wiederholt die letzte Operation, sucht also die nächste Fundstelle oder fährt mit dem Suchen und Ersetzen fort.

Back to BASIC

CTRL-K/Q Editor verlassen. Mit RUN kann man wieder in den Editor zurückkehren, ohne den Text zu verlieren.

Der Assembler: die Maschine für Maschinensprache

Sämtliche Assemblerbefehle werden wie die Editor-Kommandos über CTRL und eine weitere Taste eingegeben. Eventuell erforderliche Eingaben erwartet das Programm in der Statuszeile.

CTRL-K/X Text assemblieren. Dabei wird abgefragt:

CODE Dadurch wird festgelegt, wohin der erzeugte Maschinencode geschrieben werden soll; ob in den Speicher, auf Diskette oder nirgendwohin. Beispiele:

CODE:82DEST.O: Objektcode unter dem Namen „DEST.O" auf Diskette schreiben.

Speicherbelegung INPUT-ASS-64

$$0801 –	$2622	Assembler/Editor
$2623 –	$B6FF	Textspeicher, aktuelle Endadresse in Statuszeile
$B700 –	$BFFF	diverse Puffer, Anfang Symboltabelle
$C000 –	$CFFF	nicht benutzt
$D000 –	$DFFF	I/O-Bereich
$E000 –	$FFFF	Fortsetzung Symboltabelle

Nach einem Reset kann INPUT-ASS-64 durch SYS 2088 wieder aufgerufen werden, in der Regel ohne Textverlust.

Speicherbelegung INPUT-ASS-128

Bank 0:		
$0800 –	$09FF	div. Assembler-/Editor-Routinen
$0A00 –	$0AFF	System-Variablen C128
$0B00 –	$10FF	div. Puffer/I
$1100 –	$12FF	System-Variablen C128
$1300 –	$1BFF	nicht genutzt
$1C01 –	$3EFF	Assembler/Editor
$3F00 –	$FEFF	div. Puffer/II, Symboltabelle
$FF00 –	$FFFF	MMU/System-Variablen
Bank 1:		
$1001 -	$FEFF	Textspeicher, aktuelles Ende in Statuszeile

Beim Start des Programms wird der Adressbereich von $0000 bis $0FFF als Common Area definiert. Nach einem Reset kann INPUT-ASS-128 durch SYS 7208 wieder aufgerufen werden, in der Regel ohne Textverlust.

CODE:RETURN: keine Angabe erzeugt keinen Code.

CODE:*: Objektcode wird in den Speicher an die Adresse des ersten ORG-Befehls geschrieben. Vorsicht, nicht in den Textspeicher oder den Assembler schreiben! (siehe unten: Speicherbelegung)

LIST Ausgabegerät des Listings. Beispiele:
LIST:47: Formatiertes Listing auf den Drukker
LIST:00: dito auf Bildschirm, erfordert SYMB:30
LIST:RETURN: kein Listing ausgeben.

SYMB Ausgabegerät der Symboltabelle.
Beispiele:
SYMB:83SYM: Symboltabelle unter dem Namen „SYM" auf Diskette schreiben.

SYMB:30: Symboltabelle auf dem Bildschirm ausgeben. Anhalten/Ende wie CTRL-K/F.
SYMB:RETURN: nicht ausgeben.

CTRL-K/D Probeassemblierung ohne Ausgaben; sinnvoll für Syntax-Checks.

Labels

Labels dürfen beliebig lang sein und aus Buchstaben und Zahlen bestehen. Das erste Zeichen muß ein Buchstabe sein. Vor jeder Label-Definition muß ein Doppelpunkt stehen. Label-Definitionen dürfen auch allein in einer Zeile stehen. Mnemonics und Pseudo-OPs können nicht als Label definiert werden. Zwischen Label und folgendem Text in derselben Zeile muß ein Leerzeichen stehen. Beispiele:

```
:LABEL1 LDA #00
:LABEL2 ;Kommentar
:LABEL3
```

Mnemonics

Die Befehle werden entsprechend der normalen 6502-Notation erwartet; Ausnahmen:

- Adressierungsart Akkumulator: „ASL" statt „ASL A"! Der Quelltext muß kleingeschrieben sein.
- Adressierungsart Indirekt-X: „(AD),X" statt (AD,X)

Der Quelltext muß klein geschrieben sein.

Assemblerbefehle (Pseudo-OPs)

Diese Befehle werden in den Programmtext geschrieben und dienen als Anweisungen für den Assembler.

b Byte-Befehl. Die Zahlen beziehungsweise der Text werden als Tabelle assembliert. ASCII-Zeichen werden durch einen vorangestellten Apostroph gekennzeichnet, fortlaufende Texte können durch Anführungszeichen eingeschlossen werden. Die einzelnen Byte-Anweisungen werden durch Kommata getrennt. Entspricht .BYT, .DC.B, .ASC bei anderen Assemblern. Beispiele:

```
B 'A,'B,'C,12,10,13,0
B "INPUT 64",13,0
:SCHWARZ = 0
B SCHWARZ
```

W Word-Befehl. Assembliert 16-Bit-Worte. Die Worte werden in der 6502-üblichen Reihenfolge gespeichert, erst Low-, dann High-Byte.

Beispiel 1:

```
W $FFEA
```

erzeugt die Byte-Kombination

```
$EA, $FF
```

Beispiel 2:

```
ORG $C000
:START LDA #00
:END RTS
. . .
:TABEL W START, END
```

erzeugt an der durch Label TABEL gekennzeichneten Adresse die Byte-Kombination „$00, $C0, $02, $C0".

S NUM assembliert Null-Bytes der Anzahl „NUM" in den Text. Zum Freihalten von Datenbereichen geeignet. Nachteil: diese Null-Bytes müssen dann jeweils auch von Diskette/Kassette geladen werden. Beispiel:

```
ORG $C000
:INIT JMP BEGIN
:BUFF S $100
:BEGIN LDA #00
```

erzeugt einen Buffer ab Adresse $C003 ($C000 plus Befehlslänge des JMP-Befehls), der $100 Bytes lang ist. Label BEGIN entspricht damit der Adresse $C103.

ORG ADRESsetzt den Programmzähler des Assemblers auf die Adresse „ADRES". Die ORG-Anweisung muß zu Anfang des Programms stehen. Wird dies vergessen, kommt es zu „unerklärlichen" Branch-Errors. Der Programmzähler kann mehrmals neu gesetzt werden, der Code wird auf jeden Fall an die Adresse des ersten Programmzählers geschrieben! Alle weiteren ORG-Definitionen beeinflussen nur den logischen, nicht den physikalischen Stand des Programmzählers. Um auch den realen Stand des Programmzählers zu beeinflussen (natürlich nur vorwärts), wird der S-Befehl benutzt:

```
S $3000-*
```

setzt den Programmzähler logisch und physikalisch auf $3000.

= Zuweisung (Equate). Beispiel:

```
:PRINTC=$FFD2
```

weist dem Label PRINTC den Wert $SFFD2 zu. Entspricht EQU bei anderen Assemblern. Das Label wird nur im ersten Durchlauf gesetzt. Eine erneute Zuweisung erzeugt einen „DD-Error".

:= Mehrfachzuweisung. Beispiel:

```
:PASS := 1
```

erlaubt die (eventuell mehrfache) Neu-Definition eines Labels, das Beispiel setzt das Label PASS auf 1. Falls es dieses Label schon gibt, wird es geändert.

PAG bewirkt einen Seitenvorschub im Listing.

LIS N setzt den Listing-Mode.

lis 0: nicht ausgeben
lis 4: ausgeben, Makros nicht expandieren
lis 5: ausgeben, Makros voll ausgeben
lis 6: ausgeben, nicht erfüllten If-Teil ausgeben
lis 7: alles ausgeben

Der Listing-Mode kann innerhalb eines Programmtextes beliebig oft gewechselt werden.

TIT TEXT

TEXT wird als Titel am Anfang jeder Seite ausgedruckt. Der Titel kann mehrmals im Programm umdefiniert werden. In den Titel können auch Steuerzeichen für den Drukker eingestreut werden.

Ausdrücke

Der Assembler verarbeitet folgende Zahlentypen:

dezimal ist Vorgabe (Default)
hexadezimal Vorzeichen '$'. Beispiel: $FFFF
binär Vorzeichen '%' Beispiel: %01010111
ASCII Vorzeichen "'. Beispiel: 'a
Programmzähler Zeichen *.
Beispiel: :LB = *

Die Zahlensysteme können beliebig vermischt werden. Beispiel:

:TABEL = $1000+12

weist TABEL den Wert $100B zu.

Rechenoperationen

Der Assembler arbeitet mit vorzeichenlosen 16-Bit-Zahlen.

+ - * / : Grundrechenarten
& : logisches AND
! : logisches OR
£ : logisches EOR
= < > : Vergleiche. $FFFF=wahr, 0=falsch

Die Ausdrücke werden von links nach rechts ausgewertet. Klammern sind nicht erlaubt. Außerdem gibt es noch die Operatoren HIGH und LOW, die den ganzen Ausdruck beeinflussen. Beispiel:

:VIDEO = 1024

LDY #>NAME+10 ;High-Byte
LDX #<NAME+10 ;Low-Byte

führt zu

LDY #$04
LDX #$10

Kommentare

; Kommentar-Einleitung. Kommentare gehen bis zum Zeilenende. ';←' gibt im Listing einen Strich.

Makros

Mit Makros kann man eigene Befehle konstruieren. Die Syntax ist

1. Name des Makros
2. Makro-Befehl 'm'
3. eventuelle Parameter
4. Befehle dieses Makros
5. Endekennzeichen '/'

Das durch '/' markierte Ende der Makro-Definition muß allein in der Zeile stehen. Beispiel:

:MESSAGE M 1 ;Makro MESSAGE mit 1 Parameter
LDY #>@0 ;@0=1. Parameter
LDA #<@0
JSR $AB1E
/

Dieses so definierte Makro kann aufgerufen werden mit

MESSAGE USERTXT

und entspricht dann folgendem expandierten Text:

LDY #>USERTXT
LDA #<USERTXT
JSR $AB1E

USERTXT muß natürlich vorher definiert sein.

Ein Makro kann bis zu 10 Parameter haben, beim Aufruf muß diese Anzahl eingehalten werden. Beim Aufruf können beliebige Ausdrücke übergeben werden. Man kann Makros mehrmals aufrufen und auch

INPUT-ASS-Befehle Kurzübersicht

Alle Befehle sind zu verstehen als CTRL und gleichzeitig die angeführten Tastenkombination(en). Bei Befehlen, die aus zwei Buchstaben bestehen, kann der zweite Buchstabe auch ohne CTRL eingegeben werden.

V Insert-Modus ein/aus
P Control-Zeichen-Eingabe

Cursor-Bewegung

W Scroll up
E Zeile hoch
R Seite rauf
A Wort links
S Zeichen links
D Zeichen rechts
F Wort rechts
Z Scroll down
X Zeile runter
C Seite runter
I Tabulatorsprung
QS Zeilenanfang
QD Zeilenende
QR Textanfang
QC Textende
QB Blockbeginn
QK Blockende

Einfügen und Löschen

G Zeichen unter Cursor löschen
QY Zeichen rechts ab Cursor löschen
Y Zeile löschen
QL UNDO

Block-Handling

KA Ganzen Text als Block markieren
KB Blockanfang markieren
KK Blockende markieren
KC Block kopieren
KV Block verschieben
KY Block löschen

Statuszeile

D Aktuelle Eingabe wird Default

File-/Peripherie-Befehle

KE Disk-Befehl senden
KF Directory
KR Block laden
KS Bereich speichern (nur C64!)
KW Block schreiben
KQ Editor verlassen

Suchen und Ersetzen

J Sprung zum Label
QF Zeichenfolge suchen
QA Zeichenfolge austauschen
L Letzten Befehl wiederholen

Assembler

KX Assembleraufruf
KD Probeassemblierung

verschachteln. Ein Makro kann erst aufgerufen werden, nachdem es definiert wurde. Probleme kann es mit Labels geben:

- Alle Labels sind global, nichtssagende Bezeichner wie LOOP provozieren Double-Definition-Errors. Bewährt haben sich Bezeichnungen, bei denen die ersten zwei bis drei Zeichen angeben, zu welcher Prozedur sie gehören.
- Innerhalb von Makros führen doppelte Definitionen nicht zu einer Fehlermeldung.
- Makros mit Vorwärts-Referenzen müssen statt durch 'm' durch 'mf' definiert werden:

```
:COM MF 2
LDX 0+1
CPX 1+1
BNE COM2
LDA 0
CMP 1
:COM2
/
```

Bedingte Assemblierung

Der Programmierer hat durch diese Assembler-Option die Möglichkeit, bestimmte Programmteile nur unter festgelegten Bedingungen assemblieren zu lassen.

```
:DEBUG = 1
IF DEBUG
JSR TEST
EL
JSR NORMAL
EI
```

Wenn die Bedingung erfüllt ist (<> 0), wird der folgende Code bis EL (else) assembliert, wenn nicht, wird der Code hinter EL beziehungsweise EI (endif) assembliert. Im Beispiel wird also JSR TEST assembliert, aber nicht JSR NORMAL. IF, EL und EI müssen am Anfang der Zeile stehen. Die Bedingungen können auch verschachtelt werden.

Include-Files

Mit dem Include-Befehl können Source-Module von der Diskette assembliert werden. Beispiel:

```
IN 84INFILE
```

Der File-Name wird im selben Format wie im Editor angegeben und geht bis zum Ende der Zeile. Include-Befehle können nicht verschachtelt werden; in einem Include-File ist keine Makro-Definition möglich (aber natürlich der Aufruf eines vorher im Haupttext definierten Makros).

Phase Errors

Der Pseudo-Befehl PHC schaltet die Kontrolle auf Phase-Errors ein. Beispiel:

```
ORG $C000
PHC
LDA TEST
:TEST = 0
:END
```

Im ersten Durchlauf wird LDA als 3-Byte-Befehl assembliert, da die Variable TEST noch nicht bekannt ist. Im zweiten Pass macht der Assembler daraus einen 2-Byte-Befehl. Zero-Page-Variablen müssen also vor der Benutzung definiert werden! Mit Hilfe der bedingten Assemblierung kann man übrigens auch sehr schöne Phase-Errors zusammenbasteln . . .

Fehlermeldungen

Fehlermeldungen werden in der Statuszeile angezeigt, der Cursor wird im Text hinter den Fehler positioniert. Falls ein Fehler in einem Makro oder in einem Include-File auftritt, wird zuerst der Makro-Aufruf beziehungsweise die fehlerhafte Zeile im Include-File angezeigt, nach Drücken einer beliebigen Taste erscheint die Fehlermeldung, der Cursor ist jetzt auf die fehlerhafte Zeile im Makro oder auf den Aufruf des Include-Files gesetzt.

AM Illegale Adressierungsart (adress-mode error); Beispiel:

```
STA #8000
```

Die Adressierungsart „immediate" kann nicht Ziel eines Befehls sein.

BR Zu weiter Sprung (branch-error); Beispiele: bedingter Sprung weiter als 127 Bytes vorwärts oder -128 Bytes rückwärts; oder erste ORG-Anweisung fehlt.

DD Label doppelt definiert (double definition error); Beispiel:

```
:PRINT = $AB1E
ORG $C000
:PRINT LDX #00
```

Tu's nur einmal

Einem Symbol wurde erst (durch '=') ein Wert zugewiesen, und das gleiche Symbol wird später als Label im Programm verwandt.

MD: Falsche Makro-Definition (macro-definition-error); Beispiel:

Es wurde nicht als erstes der Name des Makros, sondern die Makro-Anweisung selbst (M) gegeben.

NO Kein Fehler

OP Illegaler Opcode (opcode-error); Beispiel:

Leerzeichen zwischen Befehl und Operand fehlt.

OV Zahl zu groß (overflow-error); Beispiel:

```
STA 80000
```

Der Operand ist größer als 65535.

PH Phase error; im zweiten Pass ergibt sich für eine Adreßberechnung ein anderer Wert als im ersten Pass – selten auftretender Fehler.

Syntax-Probleme

SN Syntax error; Beispiel:

',W' oder ',B' statt 'W' beziehungsweise 'B'.

SV Symboltabelle zu groß (symbol-table-overflow); die Symboltabelle paßt nicht mehr in den Speicher. Lösung: Text in zwei Teilen assemblieren.

TM Falsches Symbol; Beispiel:

Der erste Parameter in einem Makro wird über '@1' statt über '@0' angesprochen.

UD Label nicht definiert (undefined label); es wurde ein Label angesprochen, das im Programm nicht existiert. P. Dornier/JS

Kleinigkeiten

Toolbox

Wandlerprogramme für 'fremde' Programmtexte und einen Symboltabellen-Sortierer finden Sie in der ID-Werkstatt dieses INPUT-Specials.

SymSort dient dazu, vom Assembler auf Diskette abgelegte Symboltabellen (durch SYM:82FILE) alphabetisch zu ordnen. Es muß zunächst, wie alle Programme der Werkstatt, durch CTRL-S auf einen eigenen Datenträger überspielt werden. Von dort wird es geladen und mit RUN gestartet. Die Abfrage nach „Quelle" und „Ziel" erwartet Antworten in der INPUT-ASS-üblichen Datei-/Gerätebeschreibung, wie '82NAME' für ein Disketten-File, '47' zum Drucken und so weiter.

Die ebenfalls abgefragte „Anzahl der Labels pro Zeile" dient der sinnvollen Einstellung auf das Gerät, eine einspaltige Ausgabe ist wichtig, wenn die erzeugte Datei vom Assembler weiterverarbeitet werden soll. Achtung mit zu langen Files: symb-sort testet nicht auf Speicherüberlauf.

Mit den **Wandlern** lassen sich Source-Codes fremder Assembler in das INPUT-ASS-Format bringen. Berücksichtigt werden Profi-Ass (Data Becker) und Hypra-Ass (Markt & Technik). Beide Wandler arbeiten nicht vollautomatisch, sondern der erzeugte Text bedarf noch einiger Nacharbeit von Hand.

Die Programmtexte im BASIC-Format von Profi-Ass müssen zunächst als sequentielles File auf Diskette gelistet werden. Das geht so:

OPEN2,8,2,"NAME,S,W":CMD 2:LIST

Nach der READY-Meldung muß die Ausgabe durch PRINT#2 wieder normalisiert und das File mit CLOSE 2 geschlossen werden. Dieses sequentielle File kann man nun mit PROFI-ASS.WND bearbeiten. Die vielleicht verwirrende Abfage 'SYS 32768 ueberlesen?' hat zur Folge, wird sie mit 'Ja' beantwortet, daß das Programm im alten Text nach dem Assembler-Aufruf 'SYS 32768' sucht und erst von dort an den Text interpretiert.

Bei Hypra-Ass-Texten ist die Vorgehensweise ähnlich, mit dem Unterschied, daß die Original-Listings mit Sekundäradresse '1' zu laden sind und dann der BASIC-Anfang auf die Anfangsadresse gesetzt werden muß. Letzteres bewirkt die Befehlsfolge

POKE 43,Low-Byte Anfangsadresse
POKE 44,High-Byte Anfangsadresse.

Problem: Woher weiß der Besitzer eines Rechners ohne Floppy-Speeder die Anfangsadresse eines Programms? Antwort: MLM64plus. JS

So tun, als ob

6502-Prozessor-Simulator

Das Austesten von selbstgeschriebenen Maschinenspracheprogrammen kann zur zeitraubenden Angelegenheit werden, da nach dem Start so eines Programms niemand mehr da ist, der es kontrolliert. Mit dem 6502-Simulator können Sie Ihre Programme schon im Source-Code testen.

Dieser Simulator mit integriertem Editor war ursprünglich Teil der Assembler-Schule (ab INPUT 64, 3/87 und Special 2), hat sich aber so gut bewährt, daß wir ihn noch einmal als eigenständiges Programm veröffentlichen.

Nach dem Abspeichern auf einen eigenen Datenträger und dem Start mit RUN befinden Sie sich im Editor. Vom Editor aus gelangen Sie mit F7 in den integrierten Simulator.

Ausführliche Hinweise zur Bedienung des Editors und des Simulators sind im Programm enthalten. Sie können Sie von dort aus jeweils mit der Funktionstaste F6 aufrufen. Es wird empfohlen, diese Seiten vor der Benutzung des Programms einmal gründlich zu lesen. Besitzer eines Druckers können sie innerhalb vom INPUT-Betriebssystem auch mit CTRL-B zu Papier bringen.

Mit der RUN/STOP-Taste gelangen Sie in ein Disk-Menü, über das Programmtexte geladen und gespeichert, ein Directory gezogen und andere Peripherie-Operationen ausgeführt werden können.

Der Simulator verarbeitet beliebige, mit dem INPUT-ASS-Editor geschriebene Programmtexte; mit einer Einschränkung: Labels sind nur auf vier Stellen signifikant. Das heißt, daß beispielsweise die Verwendung von LOOP1 und LOOP2 zu einem 'double definition error' führt. JS

IMPRESSUM:

INPUT 64 Das elektronische Magazin
Special 2: 6502-Maschinensprache
Verlag Heinz Heise GmbH & Co KG
Helstorfer Straße 7 · 3000 Hannover 61
Postfach 61 0407 · 3000 Hannover 61
Telefon: (05 11) 53 52-0

Postgiroamt Hannover, Konto-Nr. 93 05-308 (BLZ 250 100 30)
Kreissparkasse Hannover, Konto-Nr. 000-01 99 68 (BLZ 250 502 99)

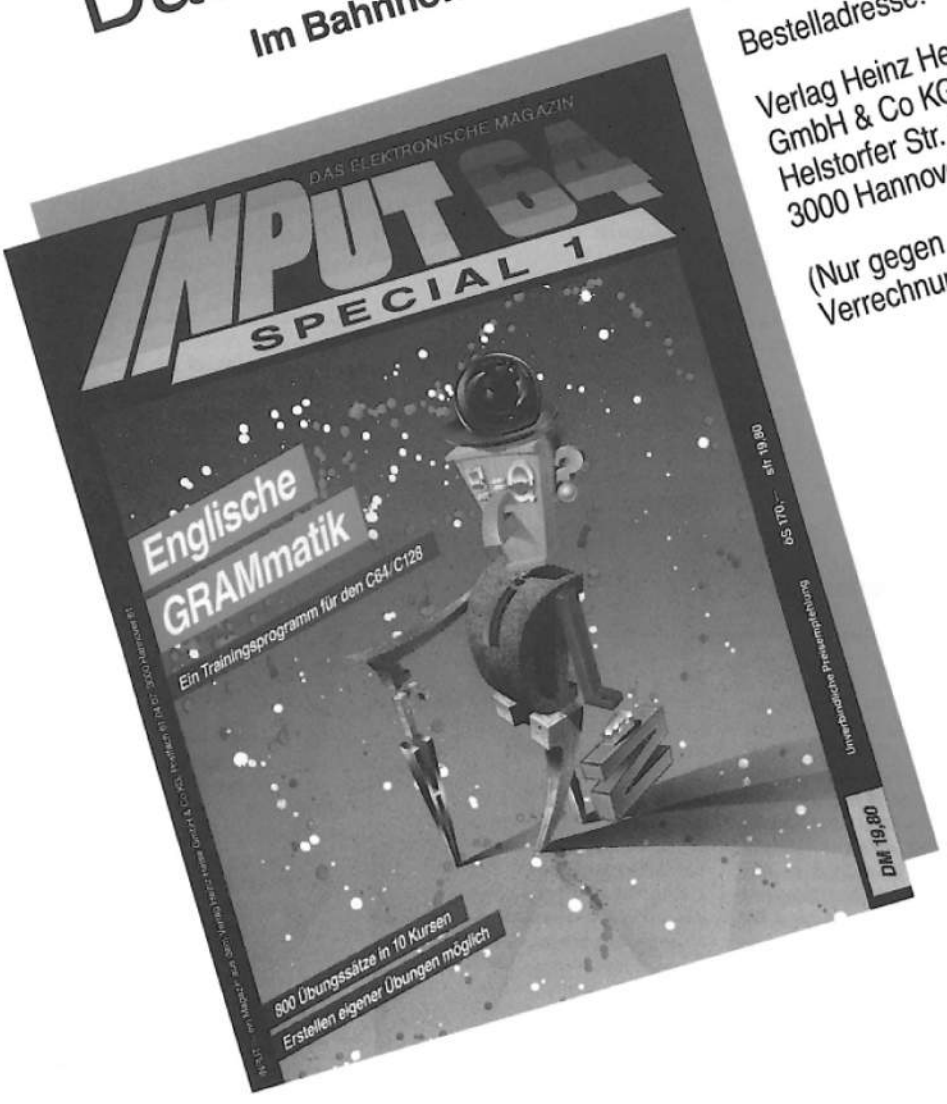
Herausgeber: Christian Heise
Redaktion: Jürgen Seeger
Vertrieb: Anita Kreutzer
Grafische Gestaltung: Wolfgang Ulber, Dirk Wollschläger, Ben Dietrich Berlin
Herstellung: Heiner Niens
Lithografie: Reprotechnik Hannover
Druck: SONOPRESS GmbH, Gütersloh
Diskettenherstellung: SONOPRESS GmbH, Gütersloh
Vertrieb (auch für Österreich, Niederlande, Luxemburg und Schweiz): Verlagsunion Zeitschriften-Vertrieb Postfach 57 07 · D-6200 Wiesbaden Telefon: (0 61 21) 2 66-0
Verantwortlich: Jürgen Seeger Helstorfer Straße 7 · 3000 Hannover 61

ISSN 0177-3771

Titelidee: **INPUT 64**
Titelillustration: M. Thiele, Dortmund

Das Lernprogramm.

Im Bahnhofsbuchhandel und direkt beim Verlag.



Bestelladresse:

Verlag Heinz Heise
GmbH & Co KG
Helstorfer Str. 7
3000 Hannover 61

(Nur gegen
Verrechnungsscheck)

HEISE

Voller Durchblick

MLM64plus:

Maschinensprache-Monitor für Floppy und Rechner

Das wahrscheinlich meistbenutzte Werkzeug von allen, die ihren C64 selbst programmieren, ist ein Maschinensprache-Monitor. Obwohl die Bezeichnung dies nicht nahelegt, ist dieses Tool auch für BASIC-Programmierer interessant. Sei es zum Speichern oder Laden beliebiger Adreßbereiche mit Sprite- und Grafikdaten, sei es für so einfache, aber praktische Kleinigkeiten wie Directory und Disk-Befehle. Dieser Monitor kann sogar rechnen und in die Floppy gucken.

Die Bezeichnung „Monitor" für dieses Programm ist bei Licht besehen krassestes Understatement. Der MLM64plus kann natürlich auch, wie jeder Monitor, Speicherinhalte lesen, verändern, laden oder abspeichern. Wie bei jedem guten Monitor stehen außerdem ein Disassembler, ein Zeilenassembler zum Eingeben von kleinen Maschinenprogrammen und ein Debugger zur Verfügung. Wie nur wenige der guten Monitore kann er auch rechnen. Und das „plus" hat er sich mit seinen umfangreichen Floppy-Optionen verdient: Directory-Funktion, Disk-Befehle senden, Sektoren lesen und schreiben, Zugriffsmöglichkeit auf das Floppy-RAM.

Das „plus" führt der MLM64plus noch aus einem weiteren Grund im Namen: im März 1985 haben wir den von Pascal Dornier aus Zollikon/Schweiz geschriebenen MLM64 veröffentlicht. Dieser Monitor ist seit über zwei Jahren tagtäglich in der Redaktion im Einsatz. In den letzten Monaten hat Hajo Schulz alle uns bekannt gewordenen kleinen „Bugs" beseitigt und über zehn neue Befehle eingebaut. Man mag einwenden, es sei nicht gerade ein Zeichen von Einfallsreichtum, ein Programm zu überarbeiten und noch einmal zu veröffentlichten. Nur: in dieses Programm sind dadurch die Erfahrungen zweijährigen professionellen Einsatzes geflossen. Und das wiegt mangelnde Phantasie allemal auf.

Speichern Sie eine der drei Monitor-Versionen (oder gleich alle drei) auf Ihre eigene Diskette ab. Geladen wird jede Version mit Sekundäradresse 1, also LOAD"NAME",8,1. Starten Sie den MLM64plus mit SYS adresse, beispielsweise mit SYS 49152 die wohl am häufigsten benutzte Version C. (Diese Beschreibung bezieht sich im folgenden bei Adreßangaben nur auf diese Version, im Kasten „Drei gute Adressen" sind die entsprechenden Werte für die anderen beiden Versionen nachzulesen.) Der Monitor meldet sich mit seinem Namen, zeigt die Inhalte der CPU-Register an und erwartet mit blinkendem Cursor Eingaben.

Erstes Training

Zur Einübung kann man beispielsweise eintippen

'm a09e

und diese Zeile mit RETURN abschließen. (Das Zeichen vor dem 'm' wird über SHIFT und 7 erreicht.) Gezeigt wird einem ein Teil des BASIC-ROMs; dergestalt daß

– zunächst ein Kürzel für die Zahlenbasis ausgegeben wird, in diesem Fall '$', also hexadezimal
– man anschließend die Adresse erfährt, ab der die folgenden Ausgaben beginnen
– dann acht Bytes in hexadezimaler Form dargestellt sind
– und am rechten Bildschirmrand diese acht Bytes noch einmal in ASCII-Darstellung erscheinen. Mit der Beispieladresse sehen Sie einige BASIC-Befehle im Klartext.

Der Monitor benutzt den Full-Screen-Editor des C64. Das heißt, man könnte jetzt mit dem Cursor auf die einzelnen Bytes wandern, eines ändern und mit RETURN diese Änderung durchführen. Das geht deswegen, weil der Monitor seine Ausgaben immer mit gültigen Eingaben beginnt. Das '$'-Zeichen ist nämlich gleichzeitig der Befehl, mit dem an eine bestimmte Adresse ein bis acht Bytes geschrieben werden können.

Eingaben über die Ausgaben

Naheliegend wäre, sich die Änderung jetzt mit einem erneuten „'m a09e" und RETURN anzusehen. Da wir es hier aber mit ROM zu tun haben, das bekanntlich nicht beschrieben werden kann, bliebe die Änderung wirkungslos. Man kann jetzt, weniger einer sinnvollen Anwendung wegen, sondern um die Bedienung des MLM64plus zu demonstrieren, zweierlei tun.

Möglichkeit 1: „t a000 a100 8000" und RETURN eingeben; dann „'m 809e" und RETURN, mit dem Cursor hoch bis auf die '45' neben der Adresse $809E; diese '45' mit '41' übertippen, mit RETURN eingeben, zwei Zeilen höher auf das „'m 809e" und dieses mit RETURN dem Monitor erneut zur Verarbeitung übergeben. Aus dem Wort 'end' ist 'and' geworden. 't' ist nämlich der Transfer-Befehl, mit dem in diesem Fall der Bereich von $A000 bis $A100 nach $8000 verschoben wurde, im RAM an der Zieladresse konnten wir dann die Veränderungen durchführen. Als erwünschter Nebeneffekt ist deutlich geworden, daß der MLM64plus mehrere Parameter durch Leerzeichen getrennt sehen will, nicht etwa durch Komma.

Möglichkeit 2: Noch einmal „'m a09e" und RETURN eingeben, mit dem Cursor auf die Adreßangabe 'a09e' gehen, diese in '809e' und die nebenstehende '45' in '41' ändern, mit RETURN eingeben, sich mit „'m 809e" und RETURN das Ergebnis ansehen. Die er-

Von Farb-, Schwarzweiß- und Maschinensprache- Monitoren

Kaum ein Begriff hat so verwirrend viele Bedeutungen wie das Wort „Monitor". Von dem computerfernen Inhalt „Studienüberwacher" mal abgesehen, trägt auch die Begriffsbestimmung im DV-Metier selbst nicht gerade zur Klarheit bei. Immerhin gibt es Farb-, Schwarzweiß- und Maschinensprache-Monitore – Hardware das eine, Software das andere.

Wenn man sich auf die Kategorie „weiche Ware" geeinigt hat, muß man sich längst nicht einig sein. Ein Monitorprogramm, ist in vielen DV-Lexika zu lesen, stellt nach dem Einschalten des Rechners grundlegende Funktionen zur Verfügung, wie Tastaturabfrage, Bildschirmausgabe, Lesen und Beschreiben von Speicherstellen. So ein Monitorprogramm zu veröffentlichen, wäre für den C64 nun absolut überflüssig: nach dem Einschalten stehen nicht nur „grundlegende Funktionen", sondern sogar ein betriebsbereites BASIC mit einem umfangreichen Betriebssystem zur Verfügung.

Doch in den letzten Jahren hat sich eine andere Definition von „Monitor" in der Praxis durchgesetzt, und darunter fällt auch der MLM64plus. Neben BASIC braucht man ein Programm, mit dem man sich den Speicher unmittelbarer als durch 'PEEK' und 'POKE' ansehen und beschreiben kann. Außerdem möchte man Byte-Folgen suchen und Speicherbereiche vergleichen und verschieben können. Praktisch ist auch die Möglichkeit, definierte Adreßbereiche laden und speichern zu können.

Und natürlich geht es um Maschinensprache. Und um das verbreitete Anfänger-Mißverständnis: "Wie kann ich Maschinenspracheprogramme LISTen?" So abwegig dem DV-Spezi diese Frage klingen mag, so naheliegend ist sie eigentlich. Denn mit dem BASIC-Interpreter, der einen nach dem Einschalten des C64 in seiner ewigen Warteschleife empfängt, kann man Maschinensprache nicht sichtbar machen. Der LIST-Befehl erkennt nur BASIC-Zeilen. Wie also Maschinensprache sichtbar machen? Mit einem Unterprogrammm der meisten Monitore, dem Disassembler. Der Disassembler liest nämlich die Inhalte des betreffenden Speicherbereichs und wandelt diese in entsprechenden Befehlstext, Mnemonics genannt, um. Beispielsweise die beiden Bytes '$A9' und '$00' in den Befehl 'LDA #00'.

Das funktionale Gegenstück ist der Assembler. Der wandelt eingegebene Befehle in hexadezimalen Code um und schreibt diesen in den Speicher. So kann man durchaus kleine Programme schreiben oder vorhandene Programme ändern; man wird also in die Lage versetzt, überhaupt Maschinenbefehle eingeben zu können.[1]

Soweit erst einmal zur Sprachregelung in Sachen Monitor. Man könnte etliche Punkte sicher noch ausführen. Aber das sollte hier eigentlich kein Maschinenspracheкurs werden. Den gab's nämlich unter dem Namen „INPUT64-Assembler-Schule" in den INPUT-Ausgaben 3 bis 8 dieses Jahres. JS

[1] Natürlich ist so ein Assembler, wie er im MLM64plus und anderen Monitoren eingebaut ist, nur ein unzureichendes Hilfsmittel. Ein „richtiger" Assembler kann wesentlich mehr. Zum Beispiel Labels verarbeiten und an beliebigen Stellen Befehle einfügen. Dafür haben wir den Macro-Assembler INPUT-ASS (Ausgabe 6/86).

ste Zeile sieht genauso aus wie unter „Möglichkeit 1" beschrieben. Man kann also beim 'S'-Befehl nicht nur bei der Byte-Ausgabe eigene Eingaben machen, sondern auch vorn bei der Adreßausgabe. Was nicht geht: rechts direkt in den ASCII-Dump schreiben.

Das Prinzip, dem Monitor seine Wünsche mitzuteilen, dürfte jetzt klar sein. Ab sofort wird auch vorausgesetzt, daß der Leser daran denkt, alle Eingaben mit RETURN abzuschließen, da sie sonst wirkungslos bleiben.

Kurz und knapp in drei Systemen

Für alle Befehle gelten, außer dem oben beschriebenen Full-Screen-Editor-Prinzip, bestimmte Regeln:

– Die Zahlenbasis ist nach dem Start des Monitors hexadezimal

– Zahlen können abgekürzt werden, so ist '$8' gleichbedeutend mit '$0008'.

– Als Trennzeichen sind ein oder mehrere Leerzeichen zulässig, zwischen Befehl und erstem Argument ist kein Trennzeichen notwendig.

– Die Kennungen für die Zahlensysteme sind $=hexadezimal, @=dezimal, %=binär und als Sonderfall

' (Zeichen über der 7)=ASCII. „$41", „@65", „%01100101" und „'a" sind also gleichbedeutend.

– Die Zahlenbasis ist umschaltbar durch Eingabe von nur '$', '@' oder '%'. Zahlen in dieser bevorzugten Basis interpretiert der Monitor auch ohne Präfix richtig. 16-Bit-Zahlen können nicht binär aus- oder eingegeben werden, im Binär-Modus werden 16-Bit-Zahlen (Adressen) hexadezimal aus- und eingegeben.

– Eingegebene Zahlen dürfen nicht größer als $FFFF (@65535) und nicht kleiner als Null sein.

– Für Befehle, die die Angabe einer Anfangs- und Endadresse verlangen, gilt: ist „Anfang" großer „Ende", so wird „Ende" als Anzahl interpretiert, beginnend bei Null.

– Alle Ausgaben können mit der CTRL-Taste verlangsamt, mit der Leertaste angehalten, mit der C=-Taste fortgeführt und mit RUN-STOP abgebrochen werden.

– Groß/Kleinschreibung ist signifikant: 'R' ist nicht gleich 'r'. Großbuchstaben beziehen sich in der Regel auf Operationen mit der Floppy.

– Falls Sie mal eine dieser Regeln mißachten oder einen falschen Befehl eingeben, zeigt Ihnen ein inverses Fragezeichen in der Eingabezeile den ungefähren Fehlerort an.

Die kleinen Unterschiede

Langjährige INPUT 64-Leser sind mit der Bedienung des Monitors wahrscheinlich schon im Schlaf vertraut. Extra für sie hier in geraffter Form die Unterschiede zwischen MLM64 und MLM64plus:

Neue dazugekommen sind die Befehle '>' (Diskbefehle, Directory), 'O', 'I' (Schreiben/Lesen von Floppy-Speicher), 'R', 'W' (Read/Write Track/Sektor), '+', '-' (Rechnen), '?' (Zahlensysteme wandeln), 'k' (Speicherkonfiguration einstellen.

Geändert hat sich die Ausgabe nach 'm', die Möglichkeit der Befehlswiederholung von 'd' und 'm', die Darstellung von wahlweise Sprite-/Zeichensatz-Mode durch '%' und die Parameter-Auswertung. Für letztere gilt bei allen Befehlen, die die Angabe von Start- und Endadresse verlangen: ist 'Ende' kleiner 'Start', wird 'Ende' als Anzahl interpretiert.

Entfallen ist die Prüfsummen-Option durch '+'. Die wesentlichste Änderung ist, daß mittlerweile die Aufrufadresse identisch mit der Startadresse ist.

Abkürzungen abgemacht

Für die Beschreibung der Befehle gelten folgende Vereinbarungen:

MLM64plus-Befehlsübersicht

$ [aa b0..b7]	Umschalten auf hexadezimal, 'POKE' Hexwert
% [aa b0..b7]	Umschalten auf binär, 'POKE' Binärwert
+ wert1 wert2	Addieren
- wert1 wert2	Subtrahieren
>[diskbefehl]	Diskbefehl senden
>$	Directory
? wert	Zahlensystemwandlung
@ [aa b0..b7]	Umschalten auf dezimal, 'POKE' Dezimalwert
a aa : befehl	Assemblermodus
b [n aa]	Breakpoint 'n' nach 'aa'
c aa ea za	Compare: Speicherbereiche vergleichen.
d [aa [ea]]	Disassemblieren
e [aa]	Einzelschrittbetrieb
f aa ea b	Fill: Speicherbereich füllen
g [aa]	Go: Programm ab 'aa' oder PC starten
h aa ea b0..bn	Hunt: nach Byte-Muster 'b0..bn' suchen
l "name" dev [aa]	Laden eines Programms
m [aa [ea]]	Memory-Dump: Speicherinhalt ausgeben
p [befehl]	Protokollieren auf Datei
q [aa]	'Quick'-Trace
r	Register ausgeben
s "name" dev aa ea	Speichern eines Adreßbereiches
t aa ea za	Transfer: Speicherbereich verschieben
v "name" dev [aa]	Verify
x	Rückkehr zum BASIC
y [aa]	Wie 'g', aber 'RTS' geht in Monitor
I floppyadr c64adrr	Liest Floppy-Speicher in Rechner
O c64adr floppyadr	Schreibt Rechner-Speicher in Floppy
R track sektor [aa]	Liest Block von Diskette
W track sektor [aa]	Schreibt Block auf Diskette

Quellen-Service

Vielleicht möchten Sie das eine oder andere im Monitor ändern oder einfach nur wissen, wie man so'was programmiert. Oder Sie haben Geschmack an Maschinensprache gefunden und möchten sich eine komplette „Werzeugkiste" zulegen, mit allem, was dazugehört.

Dann haben wir für Sie für 49,– DM (zuzüglich 3,– DM Versandkosten) eine doppelseitig bespielte Diskette mit dem in Ausgabe 6/86 veröffentlichten Macro-Assembler INPUT-ASS, diesem vorgestelltem Maschinensprache-Monitor, Library-Programmen zu I/O-Bedienung, Arithmetik und so weiter – und das alles im Source-Text.

aa: Anfangsadresse
ea: Endadresse
za: Zieladresse
b(n): Byte(s)

Parameter in eckigen Klammern ([]) sind optional, das heißt, sie können, müssen aber nicht eingegeben werden. Die folgende Übersicht ist nach Funktionsgruppen getrennt.

Sehen, suchen, ändern

In der Gruppe „Sehen, suchen, ändern" sind sozusagen die Basis-Funktionen eines Monitors zusammengefaßt; alle Befehle, mit denen man sich Speicherinhalte anschauen, diese verändern, kopieren, vergleichen und so weiter kann.

c aa ea za Die Bereiche von 'aa' bis 'ea' werden mit einem Bereich gleicher Länge ab 'za' verglichen. Alle Unterschiede werden angezeigt. Eine häufige Anwendung ist der Vergleich verschiedener Programmversionen, die man sich an verschiedene Adressen ins RAM lädt. Beispiel:

c 1000 2000 3000

f aa ea b Füllt von Anfangsadresse 'aa' bis Endadresse 'ea' den Speicher mit dem Byte 'b'. Beispiel:

f 400 700 '*

h aa ea b0 . . .bn Hunt-Befehl, zu deutsch Suchbefehl. Der angegebene Speicherbereich wird nach der durch die Bytes b0 . . .bn definierten Zeichenfolge durchsucht, alle Fundstellen werden ausgegeben. Beispiele:

h a000 b000 45 'n 'd
h a000 b000 20 d2 ff

k [b] Konfiguration für Speicherzugriff einstellen. 'b' enthält den Wert, der vor einem Speicherzugriff ins Register 1 geladen wird. Die eingestellte Konfiguration bezieht sich auf die Befehle 'm' und 'd' mit den dazugehörigen Änderungen sowie auf 'f', 't', 'c' und 'h'. Bei 't' und 'c' wird nur die Quelle beeinflußt.

'k' ohne Parameter zeigt die aktuelle Konfiguration ein. Siehe dazu auch den Kasten „Doppelt und dreifach". Ein direktes Beschreiben der Register 0 und 1 ist deswegen tabu! Das folgende Beispiel schaltet das BASIC-ROM ab und macht den „darunter" liegenden Adreßraum zugänglich:

k$36

m [aa [ea]] Memory-Dump ausgeben, 22 Zeilen zu je acht Bytes ohne Parameter; 22 Zeilen ab Adresse 'aa' oder den Bereich von 'aa' bis 'ea'. Ein vorangestelltes „" (wie schon erwähnt, das Zeichen über der '7') gibt rechts neben jeder Zeile die Bytes noch einmal in ASCII-Form aus. Steuerzeichen beziehungsweise Bildschirmcode werden als inverse Kleinbuchstaben angezeigt. Am Ende der Ausgabe wird ein 'm' in die folgende Bildschirmzeile geschrieben, so daß nur durch RETURN der m-Befehl mit den nächsten Adressen fortgesetzt wird. Beispiel:

m 80 10

t aa ea za Transfer-Befehl. Überträgt die Daten von der Anfangsadresse 'aa' bis zur Endadresse 'ea' zur Zieladresse 'za'. 'za' darf nicht zwischen 'aa' und 'ea' liegen. Beispiel:

t 400 500 600

$ [aa b0 . . .b7] Dieses beim Memory-Dump den Ausgaben vorangestellte Zeichen ist gleichzeitig der Befehl zur Umschaltung der Zahlenbasis auf hexadezimal und ein besserer 'POKE'-Befehl. Mit Parametern werden ab Adresse 'aa' die durch 'b0' bis 'b7' definierten ein bis acht Bytes geschrieben. Dieser Befehl wird in der Praxis selten zum Verändern von Speicherinhalten benutzt, meist läßt man sich durch 'm' den Speicherinhalt ausgeben und editiert dann den Dump. Beispiele:

$
$ 8000 30 31 32 33

% [aa b0..b7] Umschaltung der Zahlenbasis auf binär. Beim m-Befehl wird nur ein Byte pro Zeile ausgegeben – praktisch für die Zeichensatzdarstellung. Ein nochmaliges '%' ohne Parameter schaltet um auf die Darstellung von drei Bytes pro Zeile – so kann man sich Sprites ansehen. Wiederholtes '%' schaltet zwischen diesen beiden Darstellungsmöglichkeiten, die sich nur auf die Bildschirmausgabe beziehen, um. Alles andere wie '$'.

Ein bißchen programmieren

Die Methode, Programme direkt mit einem Monitor „einzuhacken", ist zwar endgültig out. Aber ein Disassembler und ein Zeilenassembler sind zum Analysieren und Entwickeln von Maschinenprogrammen unumgänglich.

a aa :befehl Assemblermodus. Mit diesem Befehl können Maschinenprogramme im 6502-Mnemonic-Format eingegeben werden. Nach Eingabe einer Zeile gibt der Monitor die nächste Adresse aus und bleibt im Assemblermodus, bis eine fehlerhafte Zeile oder eine Leerzeile eingegeben wird. Der Doppelpunkt ist zwingend. Mit dem d-Befehl disassemblierte Programme können editiert werden, da Hexdump, wie es dann zwischen Adresse und Doppelpunkt steht, ignoriert wird. So ist auch die Verschiebung von kurzen Programmstücken möglich, indem beim ersten Befehl vorn die Adresse verändert und dann mit RETURN durchgegangen wird. Natürlich müssen die Sprungadressen jeweils angepaßt werden. Das Leerzeichen zwischen Adresse und Doppelpunkt ist zwingend. Beispiel:

a 1000 : jmp 1000

d [aa [ea]] Disassembliert ohne Parameter 22 Zeilen ab aktuellem Programmzähler beziehungsweise von 'aa' bis 'ea'. Ausgege-

ben werden Hexdump und die üblichen 65xx-Mnemonics, illegale Opcodes ergeben drei Fragezeichen. Die Befehlswiederholung funktioniert analog zum m-Befehl. Beispiel:

d c000 c07f

Emulieren und Entwanzen

Die folgende Befehlsgruppe dient dem Testen von und der Fehlersuche in Programmen.

Beim Start dieser Programme unter Kontrolle des Monitors werden die Register mit vom Benutzer bestimmbaren Werten geladen und das so gestartete Programm ausgeführt. Der Programmierer muß selbst sicherstellen, daß keine Arbeitsadressen des Monitors oder gar der Monitor selbst überschrieben werden. Deswegen ist vor dem „Debuggen" ein Blick in den Kasten zum Thema „Speicherbelegung" dringend angeraten.

b [n aa] Der Befehl 'b' ohne Parameter zeigt die Breakpoints an. Breakpoints sind die maximal vier setzbaren „weichen" Haltepunkte, die von den Trace-Routinen (siehe 'e','q') erkannt werden. 'n' kann Werte von Null bis Drei annehmen und bezeichnet die Nummer, 'aa' die Adresse des Breakpoints. Gelöscht wird ein Breakpoint durch 'aa' gleich Null. Beispiel:

b 3 1003

e [aa] Einzelschrittbetrieb eines Programmes ab Programmzähler oder ab 'aa'. Dieses Programm wird ausgeführt, und nach jedem Befehl werden alle Register angezeigt und der Befehl disassembliert. Ein Breakpoint (siehe 'b'), ein illegaler Befehl oder die RUN/STOP-Taste führen zum Abbruch. Ein RTS-Befehl, dem kein vorheriges JSR im getesteten Programm entspricht, führt zu einem Einzelschrittbetrieb des Monitors selbst. Das geht meist nicht sehr lange gut. Beispiel:

e 2000

g [aa] Go-Befehl: das Maschinenprogramm ab 'aa' oder dem aktuellem Programmzähler starten, die Prozessor-Register werden mit den über '*' beziehungsweise 'r' gesetzten Werten geladen. Zurück in den Monitor führt nur ein BREAK-Befehl im exekutierten Programm. Mit dem Go-Befehl kann man Programme in Echtzeit testen. Dieser Befehl sollte, genauso wie 'e', 'q' und 'y', nur mit etwas Überlegung eingesetzt werden, da es naturgemäß keine (sinnvolle) Möglichkeit gibt, den Monitor oder wichtige Adressen des Monitors vor Überschreiben zu schützen. Zu beachten ist auch, daß ein RTS-Befehl, dem kein 'JSR' entspricht, sich die nächstbeste Adresse vom Stack holt und diese als Rücksprungadresse interpretiert.Beispiel:

g 1000

q [aa] Sogenanntes „Quicktrace". Das Programm ab der angegebenen Adresse, beziehungsweise bei fehlender Parameter-Angabe ab aktuellem Programmzählerstand, wird kontrolliert gestartet. Genauer gesagt, das Programm wird Befehl für Befehl emuliert, so daß auch ROM-Routinen „getraced" werden können. Ein Abbruch erfolgt durch gesetzte Breakpoints (siehe b-Befehl), einen illegalen oder den BRK-Befehl oder die RUN/STOP-Taste. Es wird als „History" ein Disassembling der letzten vier Befehle vor Programmabbruch und der aktuelle Registerstand ausgegeben. Zu auftauchenden Problemen siehe auch 'g'! Beispiel:

q a400

r Register anzeigen. Programmzähler, Register, Stackpointer und Status werden ausgegeben, letzterer immer binär. Es handelt sich dabei logischerweise nicht um Programmzähler und Register des Monitors selbst, sondern um die der mit 'e', 'q', 'g' und 'y' angesprochenen Programme. Die Adresse unter der Bezeichnung 'BRK' ist die des Break-Vektors ($0316) und bezeichnet normalerweise den Einsprung im Monitor für die Service-Routine, wenn der Prozessor auf ein Null-Byte (BRK) läuft. Alle ausgegebenen Werte kann man durch Überschreiben und RETURN ändern, da die Zeile mit '*' beginnt (siehe '*').

y [aa] Fast der gleiche Befehl wie 'g', mit dem Unterschied, daß der Stackpointer nicht gesetzt wird, so daß ein RTS in den Monitor zurückführt. Beispiel:

y a3b7

*** aa b b b b %b aa** Setzt die Register. Der Befehl ist hier nur aus Konsistenz-Gründen aufgeführt, da er allein nicht benutzt wird.

k-Wert	$8000-$9FFF	$A000-$BFFF	$C000-$CFFF	$D000-$DFFF	$E000-$EFFF
$37	evtl. Modul (ROM)	BASIC-ROM	RAM	I/O-Ports Farb-RAM	Kernal-ROM
$36		RAM		I/O-Ports Farb-RAM	Kernal-ROM
$35		RAM		I/O-Ports Farb-RAM	RAM
$34		RAM			
$33	evtl. Modul (ROM)	BASIC-ROM	RAM	RAM ROM	Kernal-

Diese RAM/ROM-Konfigurationen können durch den k-Befehl des Monitors eingestellt werden. Die Aufteilung gilt für lesende Zugriffe; geschrieben wird prinzipiell ins RAM. Ausnahme: sind die I/O-Ports eingeschaltet, werden diese beschrieben, nicht das 'darunter' liegende RAM. Für den C128 gelten alle k-Werte mit zusätzlich gesetztem Bit 6, beispielsweise $77 statt $37.

sondern statt dessen die mit 'r' ausgegebenen Register editiert werden.

Sicher sichern, locker laden

Ein paar Befehle braucht man natürlich auch, um seine Daten mit der Außenwelt auszutauschen und nicht-flüchtig auf Diskette zu lagern. Mehr noch: Direktzugriffe auf Diskette und Floppy-RAM sind erlaubt. Alle Befehle, die sich ohne Angabe einer Gerätenummer auf die Floppy beziehen, gehen von Gerätenummer 8 aus. Diese Voreinstellung ist in Anfangsadresse+5 des Monitors hinterlegt und somit veränderbar.

l "name" d [za] Lädt das Programm namens "name" vom Gerät mit der Nummer d in den Speicher, gegebenenfalls an die durch 'za' vorgegebene Adresse. Fehlt 'za', wird mit Sekundäradresse 1 geladen, also an die auf Diskette oder Kassette (8 bzw. 1) eingetragene Adresse. Nach dem Laden gibt der Monitor Anfangs- und Endadresse aus. Beispiel:

l "test" 8

Drei gute Adressen: Speicherbelegung der Monitore

Version C
$C000-$CFFF Monitor-Programm
$9F00-$9FFF Default Block Buffer
Start: SYS 49152

Version 9
$9000-$9FFF Monitor-Programm
$CF00-$CFFF Default Block Buffer
Start: SYS 36864

Version 6
$6000-$6FFF Monitor-Programm
$C000-$C0FF Default Block Buffer
Start: SYS 24576

Zero-Page-Belegung
$57, $58, $65 bis $70.
Adresse $00 sollte nicht, Adresse $01 kann nicht verändert werden, da der MLM64plus sie für den k-Befehl benutzt.

s "name" d aa ea Save-Befehl. Der Speicherbereich von 'aa' bis 'ea' wird auf Gerät 'd' geschrieben. Beispiel:

s "grafikbild" 8 @8192 @16191

v "name" d [aa] Verify. Alle Parameter wie beim l-Befehl, nur eben statt zu Laden ein Vergleich. Beispiel:

v "hase" 8 1111

p [Befehl] Protokollieren. Die Ausgaben werden auf eine mit Kanalnummer 4 geöffnete Datei geschrieben. Falls dieses File noch nicht vorhanden ist, wird OPEN4,4,7 ausgeführt, die Ausgabe also auf den Drukker geleitet. 'p' ohne Parameter bewirkt einen Zeilenvorschub auf dem Drucker. Man kann vor dem Start des Monitors ein File eröffnen (etwa: OPEN4,8,2,"PROTO,S,W") und so ein Disassemblerlisting auf Diskette schreiben. Nach dem Verlassen des Monitors ist es zwingend notwendig, das File „von Hand" zu schließen. Eine so geschriebene sequentielle Datei kann beispielsweise mit dem Macro-Assembler INPUT-ASS (Ausgabe 6/86) gelesen und weiterverarbeitet werden. Die Sekundäradresse und die Gerätenummer des Druckers stehen in Anfangsadresse+7 beziehungsweise Anfangsadresse+6 des Monitors und können so gegebenenfalls angepaßt werden. Beispiel:

pd c000

R Track Sektor [aa] Liest den durch ‚Track' und ‚Sektor' festgelegten Block an die Adresse 'aa' in den Speicher oder bei fehlender Adreßangabe in einen voreingestellten Puffer, dessen Adresse Sie dem entsprechenden Kasten („Drei gute Adressen") entnehmen können. Diese Adresse ist übrigens auch im Low-/High-Byte-Format ab Anfangsadresse+3 der Monitore hinterlegt und kann verändert werden. Nicht mit 'r' verwechseln! Beispiel:

R 12 01 8000

W Track Sektor [aa] Schreibt ab Adresse 'aa' oder ab Puffer (siehe 'R') 256 Bytes in den durch 'Track' und 'Sektor' festgelegten Block auf Diskette. Beispiel:

W 12 07 8000

=>[Disk-Befehl] Sendet einen Befehl an die Floppy. Ist dieser Befehl '$', wird das Directory angezeigt. '>' ohne Parameter liest den Fehlerkanal aus. Beispiele:

>$
>r:nameneu=namealt

O Quelladresse Zieladresse Anzahl „Output" zur Floppy. Anzahl darf zwischen 1 und 35 sein, 'Anzahl' Bytes werden ab Rechneradresse 'Quelladresse' in das Floppy-RAM ab 'Zieladresse' geschrieben. Beispiel:

O 1000 @119 1

I Quelladresse Zieladresse Anzahl „Input" von der Floppy. 'Anzahl' Bytes werden aus dem Floppy-Speicher ab 'Quelladresse' in den Rechner nach 'Zieladresse' gelesen; 'Anzahl' darf zwischen 1 und 35 sein. Nach der Ausführung des Befehls – wie auch nach 'R' – ist die Standardadresse für 'm' und 'd' auf den Anfang des geladenen Bereiches eingestellt. Beispiel:

I @119 2000 2

Individuell, aber nützlich

Manches läßt sich eben einfach nicht in vorgefertigte Rubriken pressen. Die folgenden Befehle zum Beispiel.

x Dieser Befehl bewirkt die Rückkehr zum BASIC. Kein Beispiel!

+ wert1 wert2 Gibt die Summe von 'wert1' und 'wert2' im aktuellen Zahlenformat aus. Beispiel:

+ $800 @90

- wert1 wert2 Gibt die Differenz von 'wert1' und 'wert2' aus. Beispiel:

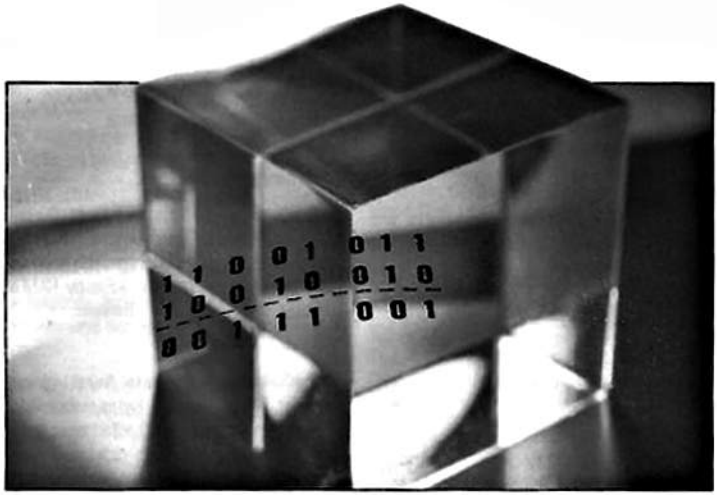
- ff fe

? wert Gibt den Wert in allen drei Zahlenformaten aus.

Rückgriffe

Die Dezimal-Umwandlungsroutine – auch schon die des 85er Monitors – stammt aus mc 3/81[1]; die Syntaxprüfung ist durch einen „Syntax-Interpreter" gelöst, wie man ihn im ROM des ZX-81 findet[2]. JS

[1] F. Laher, SED hilft bei Binär-Dezimal-Umwandlung, mc 3/81
[2] P. Dornier, ZX-81-ROM-Listing, Eigenverlag



Zurück zum Klartext

Interaktiver 6502-Re-Assembler

Ein Re-Assembler übersetzt Maschinencode in Assembler-Quelltext. Maschinenprogramme haben in der Regel die Eigenschaft, daß sie an eine bestimmte Lage im Speicher gebunden und (meist) sehr unübersichtlich sind. Wegen der festgelegten Speicheradressen ist es kaum möglich, nachträglich noch Routinen in ein schon bestehendes Maschinenprogramm einzufügen oder aus diesem herauszukürzen.

Bei einem Assembler-Quelltext bestehen dagegen diese Probleme nicht. Anstelle von festgelegten Adressen werden Symbole (sogenannte Labels) benutzt, deren Adressen vom Assembler aus den Befehlslängen berechnet werden, und die deshalb nicht an eine bestimmte Speicherlage gebunden sind. Die einzige absolute Adresse, die ein Assembler benötigt, ist die Startadresse am Anfang des Listings. Diese kann beliebig umgeändert werden. So kann durch Änderung dieser einzigen Zahl ein Programm in einem anderen Speicherbereich lauffähig gemacht werden; Programmänderungen können an beliebiger Stelle eingefügt werden.

IRAs wandelt den Maschinencode in ein Format um, das vom Assembler verarbeitet

Nur im Objectcode vorliegende Programme an eigene Bedürfnisse anzupassen, kann man eigentlich nur Leuten zumuten, die Vater und Mutter erschlagen haben. Der Interaktive Re-Assembler „IRAs" erstellt aus 6502/6510-Maschinencode editier- und kommentierbare Quelltexte für den INPUT-ASS.

werden kann. Der Quelltext, den IRAs (IRAs steht für „Interak- tiver Re-Assembler") erzeugt, ist auf den in diesem INPUT64-Special veröffentlichten Macro-Assembler INPUT-ASS zugeschnitten. Das heißt, der Text wird im Commodore-ASCII-Format abgelegt, und die Assembler-Anweisungen sind INPUT-ASS-spezifisch. Für andere Assembler muß eventuell das Textformat gewandelt und/oder die Pseudo-Ops ausgetauscht werden.

Ein Probelauf

Um Sie in das Arbeiten mit IRAs einzuführen, haben wir in das Modul, aus dem Sie den Re-Assembler selbst auf Ihren eigenen Datenträger abspeichern können, noch ein kleines Tool gepackt.

Dieses wird nicht mit CTRL und s abgespeichert, sondern über ein gesondertes Menü innerhalb des Moduls. Es ist ein sehr nützliches kleines Hilfsprogramm, ein sogenannter „Bildschirmschoner". Das heißt: wenn länger als 15 Sekunden keine Bildschirmausgabe und keine Tastatureingabe stattfindet, wird der Bildschirm ausgeschaltet. Jede normale Bildschirmausgabe oder ein beliebiger Tastendruck schaltet den Bildschirm wieder ein. Dadurch hält die Bildröhre länger (es kommt nicht mehr zu Einbrennschäden), oben-drein wird der 64er schneller. Dieses Tool wird vom eigenen Datenträger mit LOAD"SCHONER",8,1 geladen und nach einem anschließenden NEW mit SYS 52992 gestartet. Ein Neustart (nach RESET oder RUN/STOP-Restore) ist ebenfalls durch SYS 52992 möglich, der Bildschirmschoner belegt die Adressen $CF00 bis $CFA4.

Das soll hier aber nur am Rande interessieren, es ging ja um den Umgang mit einem Re-Assembler. Sie laden also zunächst IRAs von Ihrem eigenen Datenträger und starten ihn mit RUN. Das Programm meldet sich mit dem Prompt (>) und erwartet Eingaben. Durch Eingabe des Dollar-Zeichens ($) und RETURN wird das Inhaltsverzeichnis der Diskette ausgegeben. Ein Objekt-Programm wird mit F (und anschließendem RETURN, was hiermit zum letzten Mal ausdrücklich erwähnt wird) in den Speicher geladen. Beantworten Sie die Frage nach dem File-Namen mit dem Namen BILDSCHIRMSCHONER. Die Geräteadresse ist 08, IRAs unterstützt keine Kassetten-Operationen. Das Programm wird nicht an seine Startadresse, sondern an den Anfang des Arbeitsspeichers des Re-Assemblers gelesen, um Bereichsüberschneidungen zu vermeiden.

Nach dem Laden eines zu re-assemblierenden Programms ist zwingend der O-Befehl vorgegeben. Dadurch wird der Offset zwischen der momentanen Lage im Speicher und der tatsächlichen Anfangsadresse berechnet.

IRAs gibt aus:
Speicherlage: 1DC7
Offset-Adresse: CF00

Alle Ein- und Ausgaben sind natürlich hexadezimal. „Speicherlage" ist die Adresse, an

der sich der Bildschirmschoner tatsächlich befindet (für uns relativ uninteressant), „Offset-Adresse" ist die beim Laden von Diskette ermittelte Startadresse des Programms. Wir bestätigen beide Angaben mit RETURN.

Ein probeweises D (wie Disassembling) gibt Anfangs- und Endadresse des Programms vor, nach Bestätigung mit RETURN erscheint außer dem JMP-Befehl bei Adresse $CF00 wenig Sinnvolles, das B vor den einzelnen Hexzahlen steht für die Assembler-Anweisung „Byte" und sagt aus, daß sich an dieser Stelle kein Programmtext befindet. Ausführbarer Code ist erst wieder ab der Zieladresse des JMP-Befehls zu sehen ($CF26).

Versuchen wir's mit einem Memory-Dump. Das Disassembling wird mit RUN/STOP abgebrochen, nach der Eingabe von m ändern Sie die Startadresse (vor dem Komma) in CF03, die Endadresse in CF25 (einfach überschreiben). Schon besser: zu sehen ist rechts neben den Hex-Zahlen die Einschaltmeldung mit einführendem Clearhome ($93), abschließendem RETURN ($0D) und einer Null als Endekennung. So kann über den TI-Befehl der Re-Assembler wieder ein bißchen „schlauer" gemacht werden.

Über den Befehl TI (Tabellen-Input) wird eingegeben, wo sich im zu re-assemblierenden Programm Datentabellen, Programmcode und Texte befinden. (Mit Datentabellen sind die Teile eines Programms gemeint, die weder ausführbaren Maschinencode noch darstellbaren Text enthalten.) Die Abfrage nach „Tabellen löschen?" sollte vor der ersten Eingabe mit j (ja) beantwortet werden; es werden dadurch natürlich nicht die erwähnten Datentabellen selbst gelöscht, sondern die Tabelle mit den Informationen, wo überall Datentabellen zu finden sind. Die Gerätenummer 00 ist die Tastatur – stimmt in diesem Fall. Als nächstes geht es um Datentabellen, bestätigen Sie den Stern! Das heißt soviel wie „es werden keine neue Datentabellen definiert", schließlich soll ja mitgeteilt werden, wo sich Texte im Objectcode befinden. Beantworten Sie die Frage „Texte von" mit CF03, „bis" mit CF25. Durch Bestätigung des vorgegebenen Sterns mit RETURN wird der TI-Befehl beendet.

Weiter geht's mit D, diesmal wird die Anfangsadresse (vor dem Komma) durch Überschreiben in CF26 geändert. Bei der Durchsicht des Programms mit der Leertaste wird bis zum Programmende nur noch ausführbarer Code ausgegeben. Anlaß zum Nachdenken geben nur die letzten beiden Befehle (BRK und ADC $00). Dies sind zwar gültige Memonics, aber es sieht nicht nach einem sinnvollen Programm aus. Versuchen wir's mit einem Memory-Dump von $CFA2 bis $CFA4. Neben dem Hex-Dump ist den genannten Adressen kein Text zugeordnet, also handelt es sich um Datentabellen. Auch dies wird IRAs über TI mitgeteilt.

Der R-Befehl erzeugt ein Re-Assembling. Anfangs- und Endadresse werden übernommen, ebenso die Tabellierungsbreite 8, als Ausgabegerät ist zum Testen der Bildschirm sinnvoll (Gerätenummer 03). Die Ausgabe des re-assemblierten Codes kann mit der Leertaste angehalten werden, ein weiterer Druck auf die Leertaste setzt die Ausgabe fort. Bei genauerem Hinsehen können Sie feststellen, daß alle internen Sprünge „gelabelt" sind. Es heißt zum Beispiel nicht mehr „JMP $CF26", sondern „JMP LCF26", und vor dem Ziel dieses Sprungbefehls steht das Label LCF26.

Unschön sind noch die Bezüge auf Adressen außerhalb des Programms, beispielsweise JSR $AB1E oder LDA $D011. Viel aussagekräftiger wären statt dessen JSR CPRINT und LDA VIC17. „CPRINT" ist der Kurzname für die im C64-ROM installierte Routine zur Textausgabe, VIC ist der Video-Chip, dessen Basis-Adresse bekanntlich bei $D000 liegt; somit ist $D011 folgerichtig VIC17. Dem Re-Assembler wird dies beigebracht durch den LI-Befehl (Label-Input). Nach den üblichen Abfragen (Gerät und so weiter) wird dort eingegeben:

Label bei Adresse AB1E
Name des Labels: cprint

und entsprechend wird der Adresse $D011 der Name „VIC17" zugeordnet.

Ehe jetzt der re-assemblierte Quelltext auf Diskette abgelegt wird, sollten die anderen Dateien gesichert werden. Das TO-Kommando (Tabellen-Output) erlaubt, die über TI eingegebenen Werte auszugeben; auf den Bildschirm, den Drucker oder auf Diskette (Gerätenummer 08). Mit dem LO-Be-

```
          org $9000
          jmp lcf26
:text     b $93
          b "bildschirmschone"
          b "r initialisiert!"
          b $0d
          b $00
:lcf26    lda #<text
          ldy #>text
          jsr cprint
          sei
          lda #<newout
          ldy #>newout
          sta $0326
          sty $0327
          lda #<newtas
          ldy #>newtas
          sta $028f
          sty $0290
          lda #<newirq
          ldy #>newirq
          sta $0314
          sty $0315
          jsr lcf95
          cli
          rts
:newirq   lda $a1
          cmp lcfa2
          bne lcf5b
          jsr lcf88
:lcf5b    jmp $ea31
:newtas   jsr lcf95
          lda vic17
          and #$10
          bne lcf6f
          jsr lcf7b
          lda $cb
          sta $c5
:lcf6f    jmp $eb48
:newout   jsr lcf7b
          jsr lcf95
          jmp $f1ca
:lcf7b    pha
          php
          lda vic17
          ora #$10
          sta vic17
          plp
          pla
          rts
:lcf88    pha
          php
          lda vic17
          and #$ef
          sta vic17
          plp
          pla
          rts
:lcf95    pha
          php
          lda $a1
          clc
          adc #$05
          sta lcfa2
          plp
          pla
          rts
:lcfa2    b $00,$65
:cprint   = $ab1e
:vic17    = $d011
```

Der endgültige Assembler-Source-Code. Die unterstrichenen Label-Namen sind „zu Fuß" eingefügt, um das Programm in jedem Adreßbereich lauffähig zu machen.

fehl (Label-Output) können dementsprechend die über LI eingegebenen Labels ausgegeben beziehungsweise abgespeichert werden. Diese Dateien lassen sich mit dem LI- oder dem TI-Befehl wieder von Diskette einlesen. Dies ist vor allem bei der Bearbeitung längerer Programme sinnvoll, die nicht in einer Sitzung analysiert werden können. Die Dateien werden im Format des INPUT-ASS abgelegt, können also mit dessen Editor verändert werden. Es hat sich als sinnvoll erwiesen, mit dem INPUT-ASS eine Label-Datei mit den wichtigsten System-Routinen zu erstellen, die zur Re-Assemblierung eines Programms immer wieder eingelesen werden kann.

Zum Schluß unserer Beispielsitzung lassen wir das Re-Assembling auf Diskette ablegen. Nach dem R-Befehl werden Anfangs- und Endadresse mit RETURN bestätigt, ebenso die Tabellierungsbreite, als Gerätenummer wird 08 angegeben, und dann müssen Sie sich noch einen Namen für das Produkt ausdenken. Verlassen wird der Re-Assembler mit Q wie Quit. Wenn Sie sich jetzt den erzeugten Quelltext in Ihren Editor laden, ist noch ein Arbeitsschritt notwendig, um ein frei verschiebbares Programm zu erhalten. Es gibt nämlich Bezüge im Programm, die in der Form LDA #<Adresse, LDY #>Adresse vorhanden sind, zum Beispiel steht vier Zeilen nach dem Label L085A:

```
LDA #$37
LDY #$08
JSR CPRINT
```

Setzt man Low- und High-Byte zusammen, ergibt dies die Adresse $0837, und dies war im Programm exakt der Beginn der Einschaltmeldung! Die Routine CPRINT verlangt als Parameter das Low-Byte der Adresse des auszugebenden Textes im Akku, das High-Byte im Y-Register. Das wiederum kann der Re-Assembler nicht wissen. Er würde zwar LDA $0837 erkennen, aber nicht die besagte Aufteilung dieser internen Adresse. Deswegen muß der erzeugte Text noch einmal „zu Fuß" durchgegangen werden. Im beschriebenen Beispiel heißt das: ein Label vor das B $93 schreiben, etwa TEXT. (Das hätte man übrigens auch schon mit dem Re-Assembler machen können.) Die beiden Lade-Befehle müssen dann heißen:

```
LDA #<TEXT
LDY #>TEXT
```

Entsprechendes gilt für das Beschreiben der drei Vektoren im Programmteil kurz dahinter.

Interpretations-Probleme

Weitere Erläuterungen zu den Befehlen von IRAs finden Sie im Kasten „Die IRAs-Befehle". Zum gekonnten Umgang mit einem Re-Assembler gehört allerdings auch ein gewisses Verständnis der Funktionsweise solch eines Programms:

Da ein Maschinenprogramm aus einer endlosen Kette von Zahlenkombinationen besteht, die unterschiedlich interpretiert werden können (als Maschinenprogramm, als Datentabellen, als ASCII-Text), ist es für den Re-Assembler zunächst einmal wichtig zu wissen, in welcher Form der gerade zu bearbeitende Programmteil überhaupt dargestellt werden soll. (Die TI-Eingaben des Benutzers.)

Anhand dieser Angaben „weiß" der Re-Assembler nun also, an welchen Stellen reiner Programmcode liegt. Dieser wird in einem ersten Durchlauf auf Adressen untersucht, die auf eine Stelle im Programm zeigen (so-

Die IRAs Befehle

Der Editor des Re-Assemblers ist dem BASIC-Editor ähnlich. Er dient nur dazu, die einzelnen Befehle aufzurufen. Alle Tabellen können vom INPUT-ASS aus editiert werden.

R anfang, ende bedeutet Re-Assemblierung. Angaben müssen über die Start- und Endadresse gemacht werden. Werden keine Angaben gemacht, werden sinnvolle Adressen vorgegeben, falls sich ein zu re-assemblierendes Programm im Speicher befindet. Nach einem weiteren RETURN beginnt die Re-Assemblierung mit der Abfrage der Tabellierungsbreite. Diese bestimmt die Zahl der Leerzeichen vor den Assemblerbefehlen und kann Werte von Null bis 16 annehmen. Die Angabe ist dezimal.

Die Nummer des Ausgabegerätes wird hexadezimal erwartet. Erfolgt die Ausgabe auf dem Bildschirm, kann das Listing durch Drücken der Space-Taste angehalten werden. Wird bei der Re-Assemblierung der Text länger als 100 Blöcke, was keine Seltenheit ist, wird ein Include-File eröffnet. Bei sehr langen Programmen kann die Re-Assemblierung zehn Minuten und länger dauern, wobei Assemblertexte von mehreren 100 Blöcken Länge entstehen. Die Länge des erzeugten Textes hängt entscheidend von der Tabellierungsbreite ab!

D anfang,ende bedeutet Disassemblierung auf den Bildschirm.

M anfang, ende steht für Memory-Dump. Diese Funktion und der Disassembler können beispielsweise zur Feststellung von Datentabellen in geladenen Programmen verwendet werden.

F steht für „File laden" und hat die Funktion, ein zu re-assemblierendes Maschinenprogramm in den Arbeitsspeicher zu lesen. Das Programm wird nicht an seine Startadresse, sondern an den Speicheranfang geladen, um Bereichsüberschneidungen zu vermeiden.

O simuliert die richtige Speicherlage für die Bearbeitung. Der Befehl sollte nach jedem F-Befehl aufgerufen werden. Als Speicherlage wird die momentane Lage im Speicher nach dem Laden eingegeben, die Offset-Adresse ist die Adresse, an der sich das Programm normalerweise befindet (also die absolute Ladeadresse). Beide Adressen werden vorgegeben; eine Änderung kann sinnvoll sein, wenn sich im bearbeiteten Programm der Programmzähler verändert, etwa in verschobenen Programmteilen.

C sendet einen DOS-Befehl an die Floppy, nur RETURN in der Befehlszeile bedeutet Anzeigen des Disk-Status.

genannte interne Labels). Diese Adressen sollen später als Symbole dargestellt werden (der Sinn des Re-Assemblers liegt schließlich darin, alle absoluten Adressen durch Variablen zu ersetzen).

Ist die Endadresse des Programms erreicht, folgt der eigentliche Hauptteil der Re-Assemblierung, in dem der Assemblertext erzeugt wird.

Da der Assembler die jeweiligen Speicheradressen aus den Befehlslängen berechnet, ist eine Adreßangabe nicht mehr am Anfang jeder Zeile notwendig. Adressen werden lediglich am Ziel von Sprung- oder Ladebefehlen notwendig. Steht die gerade zu bearbeitenden Zeile an einer solchen Zieladresse, wird dort ein Symbol eingesetzt.

Wurde diesem Symbol kein bestimmter Name zugewiesen, wird dafür eine Variable in der Form Lxxxx (xxxx steht für eine beliebige Hex-Adresse) eingesetzt. Nun kann es aber vorkommen, daß ein Sprungbefehl diese Stelle adressiert, der erst an späterer Stelle im Programm steht. Um auch dies zu berücksichtigen, war der erste Durchlauf notwendig, der weiter nichts tat, als solche Adressen zu speichern.

Nachdem der Zeilenanfang geschrieben wurde, muß geprüft werden, in welcher Weise der nun folgende Programmcode dargestellt werden soll. Dies geschieht anhand der Festlegungen, an welchen Stellen sich Datentabellen oder Texte befinden.

Handelt es sich dabei um Daten oder Texte, werden die entsprechenden Bytes mit vorangestellten Pseudo-Opcodes in den Assemblertext geschrieben. Wurde ein Programmteil als Maschinencode definiert, wird das zu dem jeweiligen Befehl gehörende 6502-Mnemonic geschrieben, gegebenenfalls gefolgt von einem Argument. Handelt es sich dabei um eine Adresse, die in der Symboldatei gefunden wurde, so wird das entsprechende Label eingesetzt. Zusätzlich zu den internen Labels, die im ersten Durchlauf registriert wurden, werden externe Labels, die nicht im Programm liegen müssen und vom Benutzer eingegeben werden können, berücksichtigt; beispielsweise Adressen von Betriebssystemroutinen.

Diese externen Symbole erhalten ihre jeweiligen Adressen aus einer Zuweisungstabelle am Ende des Assemblerlistings.

Während die Re-Assemblierung nach diesem Grundprinzip abläuft, müssen während jeder Zeile noch einige Sonderfälle bearbeitet werden.

So kommt es zum Beispiel vor, daß die Adresse eines Sprung- oder Ladebefehls auf das zweite oder dritte Byte eines drei Byte langen Befehls zeigt. Da Symbole für Zieladressen jedoch nur am Anfang einer Zeile geschrieben werden können, würde dies zu einem Fehler führen.

Es würde auch nicht genügen, die Zieladresse absolut anzugeben, weil dann das Assemblerlisting wieder an einen bestimmten Speicherbereich gebunden wäre, und das soll ja gerade vermieden werden. In solchen Fällen wird die Zieladresse durch einen Rechenbefehl bestimmt, zum Beispiel:

LABELn = LABELx + 2.

Beim Testen von IRAs wurde versuchsweise das 64er Betriebssystem re-assembliert. Der erzeugte Source-Code war auf Anhieb ohne Nacharbeiten lauffähig. Aber: Die Qualität des erzeugten Textes hängt sehr stark von dem Umfang der eingegebenen Label-Tabelle und der Unterscheidung von Daten-, Text- und Programmbereichen ab!

Martin Friedl/JS

$ zeigt das Inhaltsverzeichnis der Diskette auf dem Bildschirm an.

Q verläßt den Re-Assembler und springt zum BASIC. Ein Neustart mit RUN ist möglich, vorhandene Informationen bleiben dabei erhalten.

LI dient zur Eingabe von internen Labels. Dies kann über Tastatur geschehen (Gerätenummer 00) oder von Diskette (Gerätenummer 08). Die Label-Dateien haben das Format des INPUT-ASS, so daß auch vorhandene Source-Texte dieses Assemblers geladen werden können, aus denen dann alle externen beziehungsweise durch das Gleichheitszeichen zugewiesenen Labels entnommen werden. Diese Symbol-Zuweisungen werden an das Ende des erzeugten Quelltextes geschrieben. Falls diese Symbole für weitere Berechnungen verwendet werden sollen, ist es sinnvoll, diese per Blockverschiebung im INPUT-ASS-Editor an den Anfang zu setzen.

LO heißt Ausgabe der Label-Datei. Dies kann auf den Bildschirm oder ein Peripherie-Gerät geschehen. Die Tabelle kann mit dem Editor des INPUT-ASS bearbeitet werden.

TI dient zur Eingabe der Darstellungsmodi der einzelnen Programmteile. Auch diese Tabellen können im Editor des INPUT-ASS geändert werden. Beim Einlesen von Diskette kommt es unter manchen DOS-Erweiterungen nach dem Laden zur Meldung „E/A-Fehler" (Ein-/Ausgabe-Fehler). Diese Meldung kann getrost ignoriert werden.

Unter „Datentabellen" können Adreßbereiche angegeben werden, die weder Programmcode noch darstellbaren Text enthalten. „Einzelne Bytes" sollten auch dann eingegeben werden, wenn an einer Stelle im Programm der BIT- Befehl zum Überspringen eines anderes Befehls benutzt wird. Beispiel:

1000 LDA #$10
1002 BIT $00A9

sollte im Assemblertext folgendermaßen aussehen:

LDA #$10
B $2C
LDA #$00

In diesem Fall muß als Adresse 1002 angegeben werden.

Bereiche, die unter „Texte" angegeben werden, erscheinen im Assemblertext als ASCII-Texte.

TO gibt die Bereichstabellen auf dem Bildschirm oder einem Ausgabegerät aus.

RUN/STOP bricht einen laufenden Befehl ab und führt zurück zur Kommando-Ebene.

Opcodes, Ausführungszeiten und Instruktionslängen

Adressierungsart	IMPLIED	IMMED.	ABSOL.	ZERO P.	Z. PAGE, X	Z. PAGE, Y	ABS. X	ABS. Y	INDIRECT	(IND, X)	(IND, Y)	RELATIVE	Beeinflußte Flags							erklärt auf Seite
Instruktionslänge	1	2	3	2	2	2	3	3	2	2	2	2								
MNEMONIC	OP n	OP n	OP n	OP n	OP n	OP n	OP n	OP n	OP n	OP n	OP n	OP n	N	V	-	B	D	I	Z	C	
ADC 1) 5)		69 2	6D 4	65 3	75 4		7D 4	79 4	72 5	61 6	71 5		N	V					Z	C	4
AND 1)		29 2	2D 4	25 3	35 4		3D 4	39 4	32 5	21 6	31 5		N						Z		13
ASL	0A 2		0E 6	06 5	16 6		1E 7						N						Z	C	12
BCC 2)												90 2									8
BCS 2)												B0 2									8
BEQ 2)												F0 2									8
BIT		89 2	2C 4	24 3	34 4		3C 4						M7	M6					Z		21
BMI 2)												30 2									8
BNE 2)												D0 2									8
BPL 2)												10 2									8
BRA 2)												80 2									–
BRK	00 7															1		1			6
BVC 2)												50 2									8
BVS 2)												70 2									8
CLC	18 2																			0	4
CLD	D8 2																0				15
CLI	58 2																	0			23
CLV	B8 2													0							48
CMP		C9 2	CD 4	C5 3	D5 4		DD 4	D9 4	D2 5	C1 6	D1 5		N						Z	C	10
CPX		E0 2	EC 4	E4 3									N						Z	C	18
CPY		C0 2	CC 4	C4 3									N						Z	C	18
DEC	3A 2		CE 6	C6 5	D6 6		DE 7						N						Z		21
DEX	CA 2												N						Z		10
DEY	88 2												N						Z		10
EOR 1)		49 2	4D 4	45 3	55 4		5D 4	59 4	52 5	41 6	51 5		N						Z		13
INC	1A 2		EE 6	E6 5	F6 6		FE 7						N						Z		21
INX	E8 2												N						Z		10
INY	C8 2												N						Z		10
JMP			4C 3						6C[4]) 6	7C[4]) 6											14
JSR			20 6																		10
LDA 1)		A9 2	AD 4	A5 3	B5 4		BD 4	B9 4	B2 5	A1 6	B1 5		N						Z		4
LDX 1)		A2 2	AE 4	A6 3		B6 4		BE 4					N						Z		10
LDY 1)		A0 2	AC 4	A4 3	B4 4		BC 4						N						Z		10
LSR	4A 2		4E 6	46 5	56 6		5E 7						0						Z	C	12
NOP	EA 2																				21
ORA		09 2	0D 4	05 3	15 4		1D 4	19 4	12 5	01 6	11 5		N						Z		13
PHA	48 3																				21
PHP	08 3																				21
PHX	DA 3																				–
PHY	5A 3																				–
PLA	68 4												N						Z		21
PLP	28 4														wiederhergestellt						21
PLX	FA 4																				–
PLY	7A 4																				–
ROL	2A 2		2E 6	26 5	36 6		3E 7						N						Z	C	13
ROR	6A 2		6E 6	66 5	76 6		7E 7						N						Z	C	13
RTI	40 6														wiederhergestellt						23
RTS	60 6																				12
SBC 1) 5)		E9 2	ED 4	E5 3	F5 4		FD 4	F9 4	F2 5	E1 6	F1 5		N	V					Z	3)	6
SEC	38 2																			1	6
SED	F8 2																1				15
SEI	78 2																	1			23
STA			8D 4	85 3	95 4		9D 5	99 5	92 5	81 6	91 6										4
STX			8E 4	86 3		96 4															12
STY			8C 4	84 3	94 4																12
STZ			9C 4	64 3	74 4		9E 5														–
TAX	AA 2												N						Z		10
TAY	A8 2												N						Z		10
TRB			1C 6	14 5									M7	M6					Z		–
TSB			0C 6	04 5									M7	M6					Z		–
TSX	BA 2												N						Z		22
TXA	8A 2												N						Z		10
TXS	9A 2																				22
TYA	98 2												N						Z		10

n = Anzahl der Taktzyklen

1) zu n 1 addieren, wenn Seitengrenze überschritten wird

2) zu n 1 addieren, wenn Verzweigung ausgeführt wird,
zu n 2 addieren, wenn bei Verzweigung Seitengrenze überschritten wird

3) Carry not = Borrow

4) Drei-Byte-Instruktion

5) NMOS-CPU: Z-Flag in Decimal Mode nicht gültig
CMOS-CPU: In Decimal Mode zu n 1 addieren

6) Drei-Byte-Instruktion: Opcode/ZP-Adresse/Offset

erweiterter Befehlssatz der CMOS-Version