Assembler-Bedienung leichtgemacht (Teil 1)

Lückenhafte Assembler-Anleitungen führen bei Anfängern häufig zu Verwirrungen. Deshalb zeigen wir am Beispiel »Hypra-Ass«, wie man einen solchen Assembler bedient.

Der Umgang mit Maschinensprache dürfte für interessierte Leser kein Problem mehr sein. Wie ein Assembler allerdings bedient wird, ist für viele immer noch ein »Böhmisches Dorf«. Deshalb zeigen wir besonders dem Anfänger, wie es gemacht wird.

In vielen Fachzeitschriften, so auch in der 64'er, stößt man häufiger auf Artikel, die in irgendeiner Form mit Maschinenprogrammierung zu tun haben. Immer mehr Leser sind auch an der Programmierung in Maschinensprache interessiert. Der C 64 läßt sich mit dem eingeschränkten V2-Basic bei weitem nicht ausnutzen. Außerdem hat ein Maschinenprogramm den Vorteil, daß es wesentlich schneller und flexibler ist, als irgendeine Hochsprache wie Basic, Pascal oder C.

Das wichtigste Instrument zum Programmieren in Maschinensprache ist ein Assembler. Es gibt sogenannte Zeilenassembler, wie sie viele Maschinensprachemonitore haben (so auch der SMON) und Mehrpass-Assembler, mit denen wir uns hier beschäftigen wollen. Zeilenassembler eignen sich nicht zum Schreiben größerer Programme. Sie sind von der Bedienung her zu umständlich.

Zuerst ein paar Worte zum Assembler selbst. Grob gesehen ist er mit einem Compiler zu vergleichen. Beide bringen einen verbalen Text, auch Quelltext oder Quellcode genannt, in eine Form, die der Computer versteht. Beide erzeugen ein lauffähiges Programm. Jeder Quelltext (ein Beispiel zeigt Listing 1) besteht aus Maschinenbefehlen, Labeln, Variablen, Formatanweisungen und sogenannten Pseudo-Opcodes.

Maschinenbefehle

Der übersetzte beziehungsweise assemblierte Quelltext besteht bis auf Tabellen (Aneinanderreihung beliebiger Zahlen) ausnahmslos aus Maschinenbefehlen. Sie steuern unmittelbar die internen Prozessorabläufe. Jeder Maschinenbefehl setzt sich aus 1, 2 oder 3 Zahlen zwischen 0 und 255 zusammen.

Die Zahl 255 ist die größte, darstellbare Zahl. Sie kommt dadurch zustande, daß der 6502-Prozessor eine 8-Bit-Struktur hat und somit acht Informationen parallel erkennt und verarbeitet. Jede dieser acht Informationen kann entweder Low = 0 oder High = 1 sein. Daraus folgt, daß insgesamt $2^8 = 256$ unterschiedliche Kombinationen möglich sind. Da die Zahl 0 auch eine Kombination ist, folgt weiter, daß sich mit acht Informationen beziehungsweise 8 Bit 255 Zahlen und die 0 darstellen lassen.

Nun aber wieder zurück zum Maschinenbefehl. Er besteht, wie gesagt, aus 1, 2 oder 3 Zahlen, auch Byte genannt. Das erste Byte ist der Operator oder das Befehls-Byte. Dieses Byte teilt dem Prozessor mit, was gemacht werden soll. Bei den beiden anderen Byte handelt es sich um Operanden, mit denen etwas gemacht wird. In einem Quelltext erscheint das Befehls-Byte, jedoch nicht in Form einer Zahl, sondern in Form von drei Buchstaben, die Opcode oder auch Mnemonic genannt weden. Beispiele für Opcodes sind LDA, STA, LDX, JSR und so weiter. Bei einem Operanden handelt es sich entweder um eine Adresse (ein oder zwei Byte), oder um ein Byte, das einen Wert darstellt und unmittelbar geladen werden kann.

»LDA 01« lädt zum Beispiel den Ak-

```
30   -.EQ ZEICHEN = $FC ;VARIABLEN VEREINBAREN
40   -.EQ TEXTLO = $FA
50   -.EQ TEXTHI = $FB
60   -.EQ CHROUT = $FFD2
70   -.EQ GET = $FFE4
80   -;
90   -.BA $9000 ;STARTADRESSE=$9000
100  -;
110  -LOOP       JSR GET          ;AUF EINGABE WARTEN
120  -           BEQ LOOP         ;WENN UNGLEICH 0 DANN WEITER
130  -;
140  -           STA ZEICHEN      ;UND MERKEN
150  -           LDX #00          ;INDEXREGISTER X MIT 0 VORBELEGEN
160  -LOOP1      LDA TASTENTAB,X;ERSTES ZEICHEN AUS TABELLE HOLEN
170  -           CMP ZEICHEN      ;UND MIT ZEICHEN VERGLEICHEN
180  -           BEQ LOOP1END     ;WENN GLEICH DANN FERTIG
190  -           CMP #"#"         ;SONST MIT ENDEZEICHEN VERGLEICHEN
200  -           BEQ LOOP         ;BEI UEBEREINSTIMMUNG WIEDER AN DEN ANFANG
210  -           INX              ;SONST X-REGISTER UM 1 ERHOEEN
220  -           BNE LOOP1        ;UND MIT NAECHSTEM ZEICHEN VERGLEICHEN
230  -;
240  -LOOP1END   LDA FUNKTABHI,X;ABHAENGIG VOM X-REGISTER HI-BYTE -1 DER ANZU-
250  -           PHA              ;SPRINGENDEN FUNKTION HOLEN UND AUF DEN STACK
260  -           LDA FUNKTABLO,X;DAS GLEICHE FUR DAS LO-BYTE
270  -           PHA
280  -           RTS              ;AN DIESER STELLE WIRD ZUR FUNKTION VERZWEIGT
290  -;
300  -TASTE1     LDX #<(TEXT1)    ;LO-BYTE STARTADRESSE DES 1.TEXTES INS X-REGISTER
310  -           LDY #>(TEXT1)    ;HI-BYTE STARTADRESSE DES 1.TEXTES INS Y-REGISTER
320  -           JMP AUSGABE      ;TEXT AUSGEBEN
330  -TASTE2     LDX #<(TEXT2)    ;DAS GLEICHE FUER DEN 2.TEXT
340  -           LDY #>(TEXT2)
350  -           JMP AUSGABE
360  -;
370  -TASTE3     RTS              ;BEI DIESER TASTE IN DEN BASICINTERPRETER VERZW.
380  -;
390  -AUSGABE    STX TEXTLO       ;LO-BYTE DES AUSGEWAEHLTEN TEXTES MERKEN
400  -           STY TEXTHI       ;DAS GLEICHE FUER DAS HI-BYTE
410  -LOOPAUS    LDY #00          ;Y-REGISTER MIT NULL VORBELEGEN
420  -           LDA (TEXTLO),Y ;UND DEN AKKU MIT DEM ERSTEN ZEICHEN LADEN
430  -           CMP #"#"         ;IST DAS ENDE SCHON ERREICHT?
440  -           BEQ LOOPAUSE     ;WENN JA DANN FERTIG
450  -           JSR CHROUT       ;SONST ZEICHEN AUSGEBEN
460  -           INC TEXTLO       ;UND DEN ZEIGER TEXTLO UND HI UM 1 ERHOEEN
470  -           BNE LOOPAUS
480  -           INC TEXTHI
490  -           JMP LOOPAUS
500  -LOOPAUSE   JMP LOOP         ;HIER WIRD WIEDER AN DEN ANFANG (EINGABE) VERZW.
510  -;
520  -TASTENTAB .TX "123#"
530  -;
540  -FUNKTABHI .BY >(TASTE1-1),>(TASTE2-1),>(TASTE3-1)
550  -FUNKTABLO .BY <(TASTE1-1),<(TASTE2-1),<(TASTE3-1)
560  -;
570  -TEXT1      .TX "DIES IST DER ERSTE TEXT"
580  -           .BY 13,"#"
590  -;
600  -TEXT2      .TX "DIES IST DER ZWEITE TEXT"
610  -           .BY 13,"#"
```

Listing 1. Beispiel-Quelltext, erstellt mit Hypra-Ass

kumulator, eine prozessorinterne Speicherstelle, mit dem Inhalt der RAM-Speicherzelle 1. Diese Befehlsart nennt sich »absolute Adressierung«.

Der Akkumulator läßt sich aber auch unmittelbar oder direkt mit einem Wert zwischen 0 und 255 laden. Dazu ist dem Wert, der geladen werden soll, ein Nummernzeichen (#) voranzustellen. Nach dem Maschinenbefehl »LDA #01« steht im Akkumulator 01 und nicht wie im ersten Beispiel, der Inhalt der RAM-Speicherzelle 01. Das Nummernzeichen wird immer dann einer Zahl vorangestellt, wenn nicht der Inhalt einer Adresse, sondern die Zahl selbst geladen werden soll. Diese Adressierungsart heißt »unmittelbare Adressierung«.

Es soll noch einmal darauf hingewiesen werden, daß die Zahl ≥ 255 sein muß.

Label und Variable

Eigentlich dürfte es die Unterscheidung Label und Variable gar nicht geben. Denn bei beiden handelt es sich um Konstante, die jeweils nur ein einziges Mal im Quelltext definiert werden dürfen. Das Arbeiten mit diesen Konstanten erleichtert das Programmieren in Maschinensprache ganz erheblich. Erst durch sie wird es möglich, Quelltextzeilen einzufügen oder zu löschen. Soll zum Beispiel ein Unterprogramm mit dem »JSR«-Befehl (entspricht dem Basic-Befehl GOSUB) aufgerufen werden, braucht man nicht in mühevoller Kleinarbeit die Startadresse des Unterprogramms ermitteln, sondern schreibt einfach vor den ersten Maschinenbefehl des Unterprogramms einen beliebigen Namen, zum Beispiel »AUSGABE«. »JSR AUSGABE« verzweigt dann in das Unterprogramm mit dem Namen »AUSGABE«. Wird der Assembler gestartet, weist er jedem Label, so auch dem Label »AUSGABE« automatisch eine absolute Adresse (Wert) zu. Dazu sind mindestens zwei Assemblerläufe notwendig. Der Grund dafür ist folgender. Stößt der Assembler zum Beispiel auf den Maschinenbefehl »JSR AUSGABE«, bevor das Unterprogramm »AUSGABE« definiert wurde, ist ihm zu diesem Zeitpunkt die absolute Adresse unbekannt. Deshalb werden im ersten Assemblerlauf, auch Pass 1 genannt, allen Labeln absolute Adressen zugeordnet, die in einer Tabelle (Symboltabelle) eingetragen werden. Erst im zweiten Assemblerlauf (Pass 2) wird der Quelltext übersetzt und die Label durch die absoluten Adressen ersetzt. Ähnlich verhält es sich mit den Variablen. Allerdings müssen sie definiert sein, bevor sie das erste Mal benutzt werden. Definiert werden sie, wie in einem Basic-Programm, mit dem Gleichheitszeichen. Zum Beispiel:

WERT = 50 oder AUSGABE = 5000

Bei einigen Assemblern ist das Gleichheitszeichen zu ersetzen durch »EQU«. Das Beispiel würde in diesem Fall lauten:

WERT EQU 50 oder AUSGABE EQU 5000

Bei Hypra-Ass wird der Definition ein ».EQ« vorgestellt, zum Beispiel: ».EQ WERT = 50«

Formatanweisungen

Wie Sie sicherlich schon bemerkt haben, sind bisher alle Zahlen im dezimalen Zahlensystem angegeben worden. Jeder Assembler akzeptiert neben dem dezimalen Zahlensystem auch Zahlen in anderen Formaten. Nämlich im »binären« und im »hexadezimalen« Zahlensystem. Das dezimale Zahlensystem eignet sich nicht besonders zum Programmieren in Maschinensprache. Der Grund dafür ist der, daß der Inhalt einer Speicherstelle oder Register kleiner gleich 255 sein muß. Wird versucht, eine Zahl größer 255 in eine Speicherstelle zu schreiben, ahndet der Computer das mit einer Fehlermeldung. Man kann eine dezimale Zahl, die größer ist als 255, nicht ohne weiteres in zwei oder drei Zahlen aufteilen, so daß sie gerade in eine Speicherstelle paßt.

Im hexadezimalen Zahlensystem ist das anders. Möchte man zum Beispiel die dezimale Zahl 258 = 0102 hexadezimal speichern, so läßt sich die hexadezimale Zahl direkt in zwei einzelne Zahlen beziehungsweise Byte aufteilen, man nennt sie oft »Low«- und »High«-Byte. In diesem Fall ist 01 das High-Byte und 02 das Low-Byte. Sie können unmittelbar hintereinander in die Speicherzellen geschrieben werden. In einem Assembler-Quelltext wird eine hexadezimale Zahl dadurch gekennzeichnet, daß ihr ein Dollarzeichen vorangestellt ist. Zum Beispiel:

LDA $CFFF = LDA 53247
LDA #$30 = LDA #48

Zusammenfassend läßt sich zum hexadezimalen Zahlensystem sagen, daß es sich besonders gut bei allen Opcodes eignet, die in irgend einer Form etwas mit Adressen zu tun haben.

Das binäre Zahlensystem, in dem nur die Ziffern 0 und 1 vorkommen, hängt unmittelbar mit der 8-Bit-Struktur des Computers zusammen. Die Zahl 255 läßt sich im binären Zahlensystem durch eine 8stellige Zahl darstellen (11111111). Dieses Zahlensystem eignet sich besonders gut bei allen logischen Operationen wie »OR«, »AND« und so weiter. Mit Hilfe der AND-Operation können bestimmte Bits gelöscht beziehungsweise isoliert werden. Mit der logischen OR-Verknüpfung lassen sich dagegen bestimmte Bits setzen. In einem Assembler-Quelltext wird eine binäre Zahl dadurch gekennzeichnet, daß ihr ein Prozentzeichen vorangestellt wird. Zum Beispiel:

AND #%00010000 isoliert das Bit 4 im Akkumulator. Das Ergebnis kann nur den Wert 0 oder $10 beziehungsweise 16 annehmen.

OR #%10000000 setzt das Bit 7 im Akkumulator. War das Bit 7 vor diesem Maschinenbefehl gesetzt, bleibt es erhalten. War das Bit nicht gesetzt, so wird zum Inhalt des Akkumulators $80 beziehungsweise dezimal 128 addiert.

Ich möchte noch darauf hinweisen, daß jede binäre Zahl bei einem unmittelbaren Befehl 8stellig ist. Ferner können nicht alle Assembler binäre Zahlen verarbeiten, so zum Beispiel der im 64'er, Ausgabe 7/85 veröffentlichte Hypra-Ass.

Pseudo-Opcodes

Neben den normalen Opcodes wie LDA, LDX, STX und so weiter, existieren noch Pseudo-Opcodes. Bei ihnen handelt es sich um Befehle, die den Assembler steuern. Auf das erzeugte Maschinenprogramm haben sie zwar eine Wirkung, erscheinen dort aber nicht. Es gibt zwei große Gruppen von Pseudo-Opcodes, punktierte, wie ».BA«, ».TX« oder ».BY«, und nicht punktierte, wie »#«, »$«, »=« und »;«. Der letzte nicht punktierte Pseudo-Opcode wurde bisher noch nicht besprochen. Er leitet einen Kommentar ein. Alles was hinter ihm steht, wird vom Assembler ignoriert und folglich nicht mit übersetzt; im Basic entspricht dem »;« die REM-Anweisung. Zum Beispiel:

LDA #$41 ; DEN BUCHSTABEN »A« IN DEN AKKUMULATOR

Der Assembler übersetzt den Maschinenbefehl LDA #$41. Der Rest der Zeile wird überlesen.

Die punktierten Pseudo-Opcodes lassen sich wieder in verschiedene Gruppen zusammenfassen:

— Assembler-Steueranweisungen
— Ausgabebefehle
— Befehle zur bedingten Assemblierung

1. Assembler-Steueranweisungen

Die Assembler-Steueranweisungen stellen die wichtigste Gruppe der Pseudo-Opcodes dar. Durch sie und durch die Verwendung von Variablen und Labeln wird erst das komfortable Arbeiten mit einem Assembler ermöglicht. Die nun folgenden Erklärungen zu den einzelnen Pseudo-Opcodes beziehen sich auf Hypra-Ass.

.BA $C000: legt die Startadresse des Maschinenprogramms fest. Hier $C000. Man kann aber auch den entsprechenden dezimalen Wert (49152) einsetzen. Zu beachten ist, daß die Startadresse vor dem ersten Maschinenbefehl stehen muß. Es ist egal, ob die Startadresse vor oder hinter der Variablendeklaration definiert wird.

.EQ LABEL = $41: Weist der Variablen oder richtiger der Konstanten »label« den Wert $41 zu.

Mit diesen beiden Pseudo-Opcodes und den Opcodes LDA, STA und RTS läßt sich schon ein kleines Maschinenprogramm erstellen. Laden und starten Sie zuerst Hypra-Ass. Anschließend sind die Zeilen entsprechend Bild 1 einzugeben:

```
10 —.BA $C000                ;STARTADRESSE NACH $C000
20 —                         ;DIES IST EINE REINE KOMMENTAR-
                             ZEILE
30 —.EQ WERT = $FF           ;IN DEN ZEILEN 30 UND 40 WERDEN
40 —.EQ ADRESSE = $9000      ;VARIABLE DEFINIERT.
50 —;
60 —      LDA #WERT          ;DER AKKU WIRD MIT DEM WERT $FF
                             GELADEN
70 —      STA ADRESSE        ;UND ANSCHLIESSEND IN DER
                             ADRESSE
80 —                         ;$9000 GESPEICHERT
90 —      RTS                ;DIESER BEFEHL BEENDET DAS
                             PROGRAMM.
```

Ein kleines Beispiel zum Umgang mit Pseudo-Opcodes

Gestartet wird der Assembler nun im Direktmodus durch den Basic-Befehl RUN. Beide Assemblerläufe (Pass 1 und Pass 2) werden automatisch hintereinander ausgeführt. Ist der Quelltext übersetzt, meldet sich Hypra-Ass mit folgender Meldung:
END OF ASSEMBLY 0:00.6
BASE = $9000 LAST BYTE AT $908C

Dieses kleine Maschinenprogramm kann nun im Direktmodus mit SYS 12*4096 gestartet werden. Der Befehl PRINT PEEK(9*4096) liefert als Ergebnis den Wert 255. Mit dem Editor-Befehl »/E« läßt sich der Quelltext LISTen. Wird in Zeile 30 der Wert $FF durch $00 ersetzt und danach der Assembler mit RUN und das Maschinenprogramm mit dem angegebenen SYS-Befehl von neuem gestartet, liefert der PRINT-Befehl als Ergebnis den Wert 0. Um wieder zu den Pseudo-Opcodes zurückzukommen, löschen Sie im Qelltext die Zeile 10 und fügen dafür die Zeile:
45 -.BA $C000
ein. Wird der Quelltext nun assembliert und das Maschinenprogramm gestartet, werden Sie keinerlei Veränderungen feststellen. Wird dagegen die Startadresse erst in Zeile 65 definiert, kann das fatale Auswirkungen haben. Es ist durchaus möglich, daß der Computer abstürzt. Aber versuchen Sie es einmal. Aus Fehlern kann man nur lernen.

Die nun folgenden drei Pseudo-Opcodes teilen dem Assembler mit, daß die in der gleichen Zeile stehenden Zeichen nicht übersetzt, sondern direkt übernommen werden sollen. Alle drei Pseudo-Opcodes sind wichtig für die Definition von Tabellen. Wichtig ist auch, daß vor jeder Tabelle ein RTS oder JMP stehen muß. Ist das nicht der Fall, interpretiert der Prozessor das erste Byte der Tabelle als Befehls-Byte. Als Ergebnis werden dann unkontrollierte Befehle ausgeführt.

.BY 255,$FF,"A": Einfügen von Bytewerten in den Quelltext. Soll eine Bytetabelle definiert werden, ist vor den Pseudo-Opcode ein Label zu setzen. Zum Beispiel:
LABEL .BY 255,$FF,"A"

.WO 1234,$FFD2,LABEL: Einfügen von 16-Bit-Adressen in den Quelltext. Die 16-Bit-Adresse wird automatisch in zwei einzelne Byte geteilt (High-Byte, Low-Byte) und anschließend im Format Low/High-Byte im Speicher abgelegt. Eine Worttabelle läßt sich genauso anlegen wie eine Bytetabelle.

.TX "dieses ist ein Beispiel": Mit diesem Pseudo-Opcode lassen sich ganze Textblöcke in den Quelltext einfügen. Prinzipiell handelt es sich bei einer Texttabelle um eine Bytetabelle, mit dem Unterschied, daß die einzelnen Byte nicht in einem Zahlenformat oder als einzelnes ASCII-Zeichen eingefügt werden, sondern eben als Textblock. Tabellen dieser Art werden häufig dazu verwendet, Texte auf dem Bildschirm auszugeben. Eine Texttabelle läßt sich genauso anlegen wie eine Bytetabelle.

Mit einem kleinen Programm (siehe Listing) soll der Umgang mit Pseudo-Opcodes gezeigt werden. Das Programm gibt auf Tastendruck einen vorgegebenen Text auf dem Bildschirm aus. In dem Beispielprogramm kommen noch einige nichtpunktierte Pseudo-Opcodes hinzu, die noch nicht erklärt wurden. Es handelt sich um das »größer«- beziehungsweise »kleiner«-Zeichen. Dieser Pseudo-Opcode dient dazu, bei einem 16-Bit Label das High-Byte (»größer«-Zeichen) beziehungsweise das Low-Byte (»kleiner«-Zeichen) zu isolieren, so daß sich vom Quelltext aus die einzelnen Byte des 16-Bit-Labels direkt, also unmittelbar, in den Akkumulator oder in das X-, Y-Register laden lassen.

In den Zeilen 240 bis 280 wird ein kleiner Programmiertrick angewendet. Zuerst wird das High-Byte der anzuspringenden Funktion auf den Stack geschrieben und anschließend das Low-Byte. Trifft das Programm nun auf einen »RTS«-Befehl, werden die beiden Byte (Low und High) in den Programmzeiger geschrieben und zur Adresse Programmzeigerinhalt + 1 verzweigt.

Aus diesem Grund wird bei der Definition der Bytetabellen in Zeile 540 und 550 von der anzuspringenden Funktion eine 1 abgezogen. Auf den ersten Blick ist es vielleicht recht unverständlich, daß die 1 nicht nur vom Low-Byte (Zeile 550) abgezogen wird, sondern auch vom High-Byte. Der Grund ist folgender. Angenommen das Label »TASTE1« entspricht der Adresse $9000. Wird nur vom Low-Byte eine 1 abgezogen, würde die »$90« für das High-Byte stehenbleiben. Das Ergebnis wäre, daß nicht das Unterprogramm bei $8FFF + 1, sondern bei $90FF + 1 in der Zeile 280 angesprungen würde.

Mit diesem Beispielprogramm möchte ich auch den ersten Teil dieses Artikels beenden. Die restlichen Pseudo-Opcodes und das Arbeiten mit Makros werde ich das nächste Mal behandeln.

(ah)