6 Programmentwicklung mit TASM

In Kapitel 2 wurden im wesentlichen die Assembleranweisungen des 8086-Prozessors vorgestellt. Die Programme ließen sich mittels DEBUG in COM-Dateien umwandeln. Deshalb wird man nach der Einarbeitung in den 8086-Befehlssatz meist auf einen Assembler umsteigen. Leider sind die Steueranweisungen zwischen A86, MASM und den Borland Turbo Assemblern (TASM) nicht ganz kompatibel. Zusätzlich enthalten die Borland Hochsprachen die Möglichkeit, Inline-Assembleranweisungen zu verarbeiten. Im vorliegenden Kapitel möchte ich auf die Steueranweisungen des Borland Turbo Assembler (TASM bis zur Version 3.0) gehen. Diese Einführung kann sicherlich nicht als Ersatz für die Originaldokumentation dienen. Besitzer des TASM steht aber ein umfangreicher Handbuchsatz zur Verfügung. Weiterhin gibt es spezielle Literatur zum TASM. Aber für die ersten Schritte sollten die folgenden Ausführungen eine gute Hilfe sein - was letztlich das Ziel dieses Buches ist.

Zur Steuerung des Assemblers müssen eine Reihe von Pseudobefehlen im Programm angegeben werden. Diese Befehle generieren keinen Code, sondern beeinflussen lediglich den Übersetzervorgang. Nachfolgend werde ich kurz auf diese Pseudobefehle (Directiven) eingehen. Weitere Hinweise finden sich in der Originaldokumentation des TASM.

Ein einfaches Assemblerprogramm zur Ausgabe des Textes *Hallo* besitzt dann folgenden Aufbau:

```
;==================================
; File:    HELLO.ASM (c) Born G.
; Version: 2.0  (TASM 2.0/3.0)
; Aufgabe: Programm zur Ausgabe der
; Meldung: "HALLO" auf dem Bildschirm.
; Das Programm ist als COM-Datei mit
; TASM zu übersetzen!!
;==================================
;
       .MODEL TINY          ; COM-Datei
       .RADIX 16            ; Hexadezimalsystem
       .CODE
       ORG 0100             ; Startadresse COM
;
HALLO: MOV AH,09            ; DOS-Display Code
       MOV DX,OFFSET TEXT   ; Textadresse
       INT 21               ; DOS-Ausgabe
;
       MOV AX,4C00          ; DOS-Exit Code
       INT 21               ; terminiere
;
```

```
; Textstring als Konstante
;
TEXT  DB "Hallo", 0A, 0D,"$"
;
END Hallo
```

Listing 6.1: HELLO.ASM im TASM-Format

Auch dieses Listing offenbart, daß die Sache wesentlich komplexer als mit DEBUG ist. Der TASM erwartet eine Reihe von zusätzlichen Steueranweisungen, deren Sinn nicht intuitiv klar wird. Um das Programm möglichst einfach zu halten, wurde hier noch das COM-File-Format verwendet, in welches die ASM-Datei zu übersetzen ist. Dieses COM-Format stammt noch aus der CP/M-Zeit und begrenzt die Größe eines Programmes auf 64 KByte. Dies ist bei unseren kleinen Programmen sicherlich kein Handikap. Vorteile bringt allerdings die Tatsache, daß DOS beim Laden eines COM-Programms alle Segmentregister mit dem gleichen Wert initialisiert (Bild 6.1).

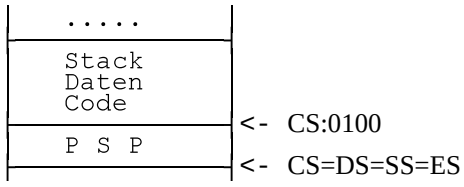

Bild 6.1: Segmentbelegung bei COM-Programmen

Ab der Adresse CS:100 beginnt der Codebereich mit den Programmanweisungen. Im gleichen Segment befinden sich auch die Daten und der Stack. Beim Zugriff auf Daten kann das Programm recht einfach aufgebaut werden. Um ein COM-Programm zu erstellen, ist obige Quelldatei mit dem Namen HELLO.ASM zu erzeugen und mit dem Befehl:

TASM HELLO.ASM,,,,

zu übersetzen. Die Kommas hinter dem Namen der Quelldatei signalisieren TASM, daß die OBJ- und Listdateien mit dem gleichen Namen, ergänzt um die entsprechende Extension, zu speichern sind. Nach dem fehlerfreien Durchlauf durch TASM muß die entstehende OBJ-Datei gelinkt werden. Die ist mit der Anweisung:

TLINK HELLO.OBJ /t

möglich. Der Schalter /T sorgt dafür, daß ein COM-Programm beim Linken erzeugt wird. Sobald das Programm fehlerfrei übersetzt und gelinkt wurde, liegt eine COM-Datei vor, die sich unter DOS ausführen läßt.

Obiges Listing enthält aber bereits eine Reihe von Steueranweisungen (Directiven), die ich nun kurz erläutern möchte.

6.1 Die Darstellung von Konstanten

Im Assemblerprogramm lassen sich Konstante innerhalb von Anweisungen (z.B. MOV AX,3FFF) eingeben. Dabei dürfen die Konstanten in verschiedenen Zahlensystemen eingegeben werden. Der TASM benutzt einige implizite Konventionen:

- Standardmäßig werden Zahlen als Dezimalwerte interpretiert (z.B. 21).
- Wird an eine Zahl der Buchstabe h angefügt, interpretiert TASM diese Zahl als Hexadezimalwert (z.B. 3FFFh). Eine Hexzahl muß immer mit den Ziffern 0..9 beginnen. Falls die erste Ziffer zwischen A und F liegt, ist eine 0 voranzustellen (z.B. C000H wird zu 0C000H).
- Binärzahlen sind durch einen angehängten Buchstaben b zu markieren. Dann sind nur die Ziffern 0 und 1 innerhalb der Zahl erlaubt.
- Dezimalzahlen lassen sich explizit durch ein angehängtes t (z.B: 100t) darstellen.
- Mit dem Buchstaben o hinter der letzten Ziffer werden Oktalzahlen markiert.

Die Buchstaben für die Zahlenbasis können sowohl in Groß- als auch in Kleinbuchstaben angegeben werden.

6.2 Die RADIX-Directive

Die obigen impliziten Definitionen für die Interpretation der Zahlenbasis ist mit zusätzlichem Aufwand verbunden. Deshalb verzichten viele Programmierer auf den angehängten Buchstaben zur Markierung der Zahlenbasis. Hier läßt sich mit dem RADIX-Kommando die eingestellte Zahlenbasis überschreiben. Die Anweisung besteht aus dem Schlüsselwort und einer Dezimalzahl, die die gewünschte Zahlenbasis spezifiziert: RADIX xx. Die folgenden Anweisungen verdeutlichen die Anwendung des Befehls:

```
RADIX 10     ; alle Werte als Dezimalzahlen lesen
RADIX 16     ; alle Werte als Hexadezimalzahlen
RADIX 2      ; alle Werte als Binärzahlen lesen
```

Dem RADIX-Kommando ist ein Punkt voranzustellen. Ich habe mir angewöhnt in allen Assemblerprogrammen mit Hexadezimalzahlen zu arbeiten. Deshalb steht die RADIX-Directive auch zu Beginn eines jeden Programmes, um die Zahlenbasis 16 zu vereinbaren.

6.3 Definition von Datenbereichen

Häufig möchte man Variable innerhalb eines Programms anlegen. Dabei ist für jeden Eintrag genügend Speicher vorzusehen.

6.3.1 Datendefinitionen mit DB, DW, DD, DQ und DT

Zur Definition von Variablen kennt der TASM - wie MASM - die Pseudoanweisungen:

```
DB     (erzeugt ein Data Byte)
DW     (erzeugt ein Data Word)
DD     (erzeugt ein Double Data Word = 4 Byte)
DF,DP (erzeugt ein FAR Data Word = 6 Byte)
DQ     (erzeugt ein Quad Data Word = 8 Byte)
DT     (erzeugt 10 Data Byte)
```

Die DQ-Anweisung erzeugt 8 Byte, welche zur Speicherung von 8087-Werten gebraucht werden. Die DT-Anweisung erzeugt eine Variable mit 10 Byte. Vor dem Pseudobefehl kann der Name der Variablen stehen:

```
Zahlen DB 1,2,3,4,5,6
Wort   DW 0FFFFH
```

Über den Namen läßt sich später auf die Daten zurückgreifen (z.B. MOV DX,OFFSET Text). Beachten Sie, daß hinter dem Variablen kein Doppelpunkt steht.

Soll ein Textstring definiert werden, kann dies in älteren TASM-Versionen (und zumindest bei den meisten Assemblern) mit dem Befehl:

```
String  DB "Dies ist ein String", 0A, 0D,"$"
```

erfolgen. Doch nun zurück zu unserem ersten TASM-Programm. Mit dem Befehl:

```
TEXT  DB "Hallo", 0A, 0D,"$"
```

wird ein Datenbereich, bestehend aus mehreren Bytes vereinbart, der sich zusätzlich über den Namen Text ansprechen läßt. Der Assembler initialisiert den Text dann mit den angegebenen Startwerten. Konstante, die mit Werten initialisiert werden, sollten im Codesegment gespeichert werden. Variable, deren Inhalt vom Programm verändert wird, sind dagegen im Datensegment abzulegen. Bezüglich der Definition der Segmente sei auf den Abschnitt *Die Segment Anweisung* und die folgenden Beispielprogramme verwiesen.

6.4 Der DUP-Operator

Um größere Datenbereiche mit dem gleichen Wert zu initialisieren, existiert das Schlüsselwort DUP. Soll zum Beispiel ein Puffer mit 5 Byte angelegt werden, müßte die DB-Directive folgendes Aussehen haben:

```
Buffer DB 00, 00, 00, 00, 00
```

Bei diesem Beispiel läßt sich die Definition noch leicht als Quellzeile formulieren. Was ist aber, falls der Puffer 256 Zeichen umfassen soll? Es macht wohl keinen Sinn hier 256 Byte mit dem Wert 0 im Quellprogramm aufzunehmen. Die meisten Assembler bieten aber die Möglichkeit einen Datenbereich mit n Byte zu reservieren und gegebenenfalls mit Startwerten zu belegen. Um einen Puffer mit 32 Byte Länge zu vereinbaren ist folgende Anweisung möglich:

```
Buffer DB 20 DUP (0)
```

Beachten Sie dabei, daß die Längenangabe hier als Hexzahl (20H = 32) erfolgt. Der Puffer wird mit den Werten 00H initialisiert. Um alle Bytes auf den Wert FFH zu setzen, wäre die Anweisung:

```
Buffer DB 20 DUP (FF)
```

erforderlich. Das bedeutet:

- Die Zahl vor dem DUP-Operator gibt die Zahl der zu wiederholenden Elemente wieder. Beim DB-Operator sind dies n Byte, beim WORD-Operator n Worte.
- Nach dem DUP-Operator folgt der Initialisierungswert in Klammern.

Soll eine Variable nicht initialisiert werden, ist an Stelle des Wertes ein Fragezeichen (?) einzusetzen.

```
Buffer DB 20 DUP (?)
```

Der Assembler reserviert dann einen entsprechenden Speicherbereich für die Variable.

ASCII-Strings lassen sich einfach in Hochkommas einschließen und im Anschluß an eine DB-Anweisung eingeben:

```
Text DB 'Die ist ein Text',0D,0A
```

In obigem Beispiel werden Textzeichen und Hexbytes gemischt. Zur Wahrung der Kompatibilität dürfen für Textkonstante auch Hochkommas verwendet werden. Es ist

aber auf die unterschiedliche Speicherung von Bytes und Worten zu achten. Die Anweisungen:

```
Text1  DB 'AB'     (speichert LOW 41H, High 42H)
Text2  DW 'AB'     (speichert 42H, 41H)
```

legen die Bytes in unterschiedlichen Speicherzellen ab. Im ersten Fall steht der Wert 'A' auf dem unteren Byte, daran schließt sich der Buchstabe 'B' an. Bei der Abspeicherung von Worten ist der linke Buchstabe dem höheren Byte zugeordnet. Mit der Anweisung:

```
Zeiger DW FFFF0000
```

lassen sich 32-Bit-Werte (FAR Pointer) definieren.

Der DUP-Operator läßt sich übrigens auch schachteln. Die Anweisung:

```
Buffer DB 5 DUP ( 5 DUP (5 DUP (1)))
```

legt einen Puffer von 5 * 5 * 5 (also 125 Byte) an, der mit den Werten 01H gefüllt wird.

Achtung: Beachten Sie aber, daß der TASM standardmäßig alle Werte als Dezimalzahlen interpretiert. Die Angaben in einer DUP-Anweisung, oder der Initialisierungswert, werden dann als Dezimalzahl ausgewertet.

```
Buffer 256 DUP (23)
```

Legt einen Puffer von 256 Byte an, der mit dem Wert 23 (dezimal) gefüllt wird. Da in der Regel bei der Assemblerprogrammierung mit Hexadezimalwerten gearbeitet wird (denken Sie nur an den INT 21 (hexadezimal)), verwende ich in meinen Programmen die RADIX-Anweisung zur Einstellung der Zahlenbasis 16. Wird dies vergessen, kommt es schnell zu Fehlern. Wer gibt schon in seinen Programmen den Befehl *INT 21H* ein. Wird die RADIX-Directive nicht verwendet, müßte der DOS-INT 21-Aufruf mit *INT 33* (dezimal) kodiert werden.

Der LENGTHOF-Operator

Häufig ist es innerhalb eines Assemblerprogramms erforderlich die Länge einer Datenstruktur zu ermitteln. Wurde ein String zum Beispiel mit:

```
Text DB "Hallo dies ist ein Text"
```

vereinbart, läßt sich dieser Text Byte für Byte bearbeiten. Der TASM bietet keine Directive LENGTHOF wie MASM zur Abfrage der Zahl der Elemente der Variablen. Mit:

```
Text DB "Hallo dies ist ein Text"
Ende LABEL WORD
LENGTHOF DW (Ende - Text)
```

wird jedoch eine Pseudokonstante LENGTHOF definiert, in die TASM die Stringlänge ablegt. Mit:

```
MOV CX, LENGTHOF Text
```

wird der Wert dann als Konstante dem Register CX zugewiesen. Der Befehl ist im Prinzip als MOV CX,23T zu interpretieren, wobei der Assembler automatisch den Wert 23T einsetzt.

6.4.2 Die Definition von Strukturen

Mit der Anweisung STRUC lassen sich mehrere Variable oder Konstante zu kompakteren Datenstrukturen zusammenfassen. Die Directive erzwingt, daß die Variable zusammen abgespeichert werden und sich anschließend durch die Zeigerregister BX, BP, DI oder SI über das erste Element adressieren lassen. Die indirekte Adressierung über [BX+SI+Konst.] eignet sich gut zur Bearbeitung solcher Strukturen. Die Definition:

```
Buffer STRUC
 Len   DB 80              ; Länge Puffer
 Count DB 00              ; Zeichenzahl
 Buf   DB 80 DUP (?)      ; Bufferbereich
Buffer ENDS
```

legt eine Datenstruktur für einen Puffer an, in der das erste Byte die Pufferlänge enthält. Im zweiten Byte ist die Zahl der im Puffer befindlichen Zeichen definiert. Daran schließt sich ein Pufferbereich von 127 Byte an. Die Definition der Struktur wird mit einer ENDS-Directive abgeschlossen.

6.4.3 Die MODEL-Directive

Zu Beginn des Programmes läßt sich dem Assembler mitteilen, welches Speichermodell gewählt wurde. Abgeleitet von den 8086-Speicherstruktur existierten verschiedene Optionen:

- TINY veranlaßt, daß Code, Daten und Stack zusammen in einem 64-KByte-Block vereint werden. Dies ist bei COM-Programmen erforderlich. Code und Zugriffe auf Daten werden dabei mit NEAR-Adresszeigern ausgeführt.

- SMALL bewirkt, daß ein 64-KByte-Codesegment und ein Datensegment gleicher Größe definiert wird. Dies setzt aber voraus, daß das Programm als EXE-File gelinkt wird.

- MEDIUM unterstützt mehrere Codesegmente und ein 64 KByte großes Datensegment. Bei dem COMPACT-Modell werden dagegen mehrere Datensegmente, aber nur ein 64-KByte-Codesegment zugelassen.

- LARGE erlaubt mehrere Daten- und mehrere Codesegmente. Hier erfolgen die Zugriffe auf Code und Daten grundsätzlich mit FAR-Zeigern.

Der TASM unterstützt weitere MODEL-Vereinbarungen, auf die ich an dieser Stelle aber nicht eingehen möchte. Unser Beispielprogramm enthält die TINY-Anweisung, damit sich das Programm in eine COM-Datei verwandeln läßt.

6.5 Die Segmentanweisung

Dem Assembler muß ebenfalls mitgeteilt werden, in welchem Segment er die Codes oder die Daten ablegen soll. Ein übersetztes Programm besteht minimal aus einem Code- und einem Datensegment (Ausnahme COM-Programme, die nur ein Segment besitzen). Bei der Erzeugung der .OBJ- und .COM-Files benötigt der TASM zusätzliche Informationen zur Generierung der Segmente (Reihenfolge, Lage, etc.). Hierzu sind mehrere Operatoren zugelassen:

- CODE markiert den Beginn eines Programmbereiches mit ausführbaren Anweisungen. Neben den Programmanweisungen können hier auch Konstante oder initialisierte Daten abgelegt werden. Interessant ist dies vor allem bei der Erzeugung von .OBJ-Files, da der Linker die Segmente später zusammenfaßt. Im TINY-Modell ist nur ein Codebereich erlaubt.

- DATA signalisiert, daß ein Datenbereich folgt. Anweisungen wie DB, DW und DQ erzeugen solche Variable. Häufig findet man die Datendefinition zu Beginn eines Assemblerprogramms. Mit der ORG-Anweisung läßt sich die Lage des Datenbereiches explizit festlegen. In den Beispielprogrammen finden sich die Definitionen der Datenbereiche häufig am Programmende hinter dem Code. Dadurch legt der Assembler automatisch die Lage des Datenbereiches korrekt fest. TASM generiert dann Anweisungen, die später im Datensegment des Programmes abgelegt werden. Obiges Beispielprogramm verzichtet auf ein Datensegment, da keine Variablen zu bearbeiten sind.

- STACK erzeugt ein Segment, welches der Linker für den Stack reserviert. In obigem Beispiel wurde auf den Stack verzichtet, da ein COM-Programm den Stack im oberen Speicherbereich des Segmentes anlegt. Mit .STACK 200H wird automatisch 512 Byte Stack reserviert.

Weitere Beispiele für die Verwendung dieser Segmentdefinitionen finden sich in den folgenden Abschnitten.

6.5.1 Die ORG-Anweisung

Mit dieser Anweisung lassen sich Daten und Programmcode an absoluten Adressen im aktuellen Segment speichern. Bei OBJ-Dateien darf die Lage der Code- und Datenbereiche nicht festgelegt werden, da dies Aufgabe des Linkers ist. Bei COM-Programmen müssen dagegen die Adressen im Programm definiert werden. Hierzu dient die ORG-Anweisung, mit der sich die Adresse des nächsten Befehls definieren läßt. Mit ORG 100 wird TASM angewiesen, die nächste Anweisung ab CS:0100 zu assemblieren. Der Parameter einer ORG-Anweisung muß entweder eine Konstante oder ein Ausdruck sein. Eine Verwendung eines symbolischen Namens ist ebenfalls möglich, sofern keine Vorwärtsreferenzen dabei notwendig werden. Bei der Assemblierung einer COM-Datei generiert der TASM den Code auch ohne die ORG 100 Anweisung ab CS:0100. In den Beispielprogrammen wird die ORG-Anweisung jedoch verwendet. Allgemein bleibt festzuhalten, daß der ORG-Befehl nur selten im Programm auftreten sollte. Andernfalls besteht die Gefahr, daß sich Codebereiche bei späteren Programmerweiterungen überlappen und damit zu Fehlern führen.

Die ORG-Anweisungen sind demnach nur bei COM-Programmen erlaubt und sinnvoll. Obiges Programm enthält z.B. eine ORG-Anweisung, so daß der Code ab CS:100H beginnt.

6.6 Der OFFSET Operator

Beim Zugriff auf Speicherzellen (z.B: MOV AX, Buffer) sind zwei Fälle zu unterscheiden. Einmal kann der Inhalt einer Konstanten *Buffer* gemeint sein. Andererseits besteht die Möglichkeit, die Adresse der Variablen *Buffer* in AX zu lesen. Um die Konstruktion eindeutig zu gestalten, existiert die Anweisung OFFSET, die in den Beispielprogrammen verwendet wird. Mit:

MOV AX, OFFSET Buffer

wird die (Offset) Adresse der Variablen oder Konstanten *Buffer* durch den Assembler berechnet und in das Register AX geladen.

Die Segmentadresse läßt sich übrigens mit dem Operator:

MOV BX, SEG Buffer

ermitteln.

Um den Inhalt der Variablen Buffer zu laden, ist die indirekte Befehlsform (z.B. MOV AX,[Buffer]) zu nutzen.

Programmbeispiel

Zur Auflockerung möchte ich an dieser Stelle ein weiteres Programmbeispiel in TASM einfügen. Es handelt sich um das Programm zur Vertauschung der beiden parallelen Schnittstellen LPT1 und LPT2. In TASM besitzt das Programm folgendes Format:

```
;=================================
; File: SWAP.ASM   (c) Born G. V 1.0
; Programm zur Vertauschung von LPT1
; und LPT2. Programm als COM-Datei
; mit TASM übersetzen!!
;=================================
;
       .MODEL TINY          ; COM-Datei
       .RADIX 16            ; Hexadezimalsystem
       .CODE
       ORG 0100             ; Startadresse COM
;
SWAP:  MOV AH,09            ; DOS-Display Code
       MOV DX,OFFSET TEXT   ; Textadresse
       INT 21               ; DOS-Ausgabe
;
       MOV AX,0000          ; ES auf Segm. 0000
       MOV ES,AX            ; setzen
       MOV AX,ES:[0408]     ; Portadresse LPT1
       MOV BX,ES:[040A]     ; Portadresse LPT2
       XCHG AX,BX           ; Swap Adressen
       MOV ES:[0408],AX     ; store Portadressen
       MOV ES:[040A],BX     ;
;
       MOV AX,4C00          ; DOS-Exit Code
       INT 21               ; terminiere
;
       .DATA
;
; definiere Textstring im Datenbereich
;
TEXT  DB 'SWAP LPT1 <-> LPT2  (c) Born G.',
0D,0A,'$'
;
END Swap
```

Listing 6.2: SWAP.ASM

Das Programm wird wieder als COM-Datei übersetzt. Als Erweiterung zur Version aus Kapitel 2 gibt das Programm beim Aufruf noch eine Meldung an den Benutzer aus. Die Textkonstante wird in diesem Beispiel explizit im Datensegment abgelegt. Die Anweisung .DATA definiert den Beginn des Segmentes. Auch hier ist wieder eine Besonderheit bei Verwendung des Segment-Override-Operators zu beachten.

6.7 Die Segment-Override-Anweisung

In Kapitel 2 wurde die Adressierung der 8086-Befehle besprochen. Standardmäßig greift der Prozessor auf das Datensegment zu. Bei Adressierung über das BP-Register finden sich die Daten im Stacksegment. Mit der Segment-Override-Anweisung läßt sich die Standardeinstellung für die Segmentierung für den folgenden Befehl aussetzen. In DEBUG muß der Segmentdescriptor vor dem Befehl stehen:

```
ES:MOV AX,[0408]    ; Portadresse LPT1
ES:MOV BX,[040A]    ; Portadresse LPT2
```

Der TASM generiert bei diesem Konstrukt eine Fehlermeldung, erwartet er doch den Segmentdescriptor vor der eckigen Klammer.

```
MOV AX,ES:[0408]    ; Portadresse LPT1
MOV BX,ES:[040A]    ; Portadresse LPT2
```

Sollte der Assembler einen Fehler in einer Anweisung mit Segment-Override melden, prüfen Sie bitte zuerst, ob obige Konventionen erfüllt sind.

Programmbeispiel

Als nächstes Beispiel möchte ich das Programm zur Abschaltung der *NumLock*-Taste in der Version für den TASM vorstellen. Das Programm besitzt folgenden Aufbau:

```
;=================================
; File: NUMOFF.ASM   (c) Born G.
; Ausschalten der NUM-Lock-Taste.
; Programm als COM-Datei mit TASM
; übersetzen!!
;=================================
;
       .MODEL TINY          ; COM-Datei
       .RADIX 16            ; Hexadezimalsystem
;
;      Programmkonstanten definieren
;
       SEG0    EQU  0000    ; 1. Segment
       Key     EQU  0DF     ; NumLock Key
       Key_Adr EQU  0417    ; Adresse BIOS
       Exit    EQU  4C00    ; DOS-Exit
       DOS_Txt EQU  09      ; DOS-Text
;
       .CODE
       ORG 0100             ; Startadresse COM
;
NUMOFF:MOV AH,DOS_Txt       ; DOS-Display Code
       MOV DX,OFFSET TEXT   ; Textadresse
       INT 21               ; DOS-Ausgabe
;
```

```
       MOV AX,SEG0          ; ES auf BIOS-Segm.
       MOV ES,AX            ; setzen
; Bit der NumLock-Taste ausblenden
       AND BYTE PTR ES:[Key_Adr],Key
;
       MOV AX,Exit          ; DOS-Exit Code
       INT 21               ; terminiere
;
       .DATA
;
; definiere Textstring im Datenbereich
TEXT  DB 'NUMOFF  (c) Born G.',0D,0A,'$'
;
END NUMOFF
```

Listing 6.3: NUMOFF.ASM

Auch hier ergibt sich wieder ein neuer Aspekt. Innerhalb des Programmes werden verschiedene Konstanten verwendet (z.B: MOV AH,09 für die Textausgabe). Ändert sich der Wert einer Konstanten, ist gerade bei größeren Programmen die Modifikation sehr fehlerträchtig. Alle Anweisungen, in denen die Konstante vorkommt, müssen gesucht und modifiziert werden. Mit der EQU-Directive läßt sich hier eine bessere Lösung finden.

6.8 Die EQU Directive

Mit EQU läßt sich einem Namen eine Konstante, ein Ausdruck, etc. zuweisen. Immer wenn im Verlauf des Programmes dann dieser Name auftaucht, ersetzt der Assembler diesen Namen durch den mit EQU zugewiesenen Wert. Statt der Anweisung: MOV AX,0FFFF läßt sich das Programm wesentlich transparenter mit der Sequenz:

```
True  EQU 0FFFF      ; Konstante True
False EQU 0          ; Konstante False
.
MOV AX,True          ; AX Init
```

gestalten. Ähnlich lassen sich Masken, Pufferlängen, etc. mit symbolischen Namen belegen und per EQU definieren. Im Programm taucht dann nur noch der symbolische Name auf. Werden die EQU-Anweisungen im Programmkopf definiert, lassen sich die Programme sehr transparent gestalten, da sich diese EQU's leicht ändern lassen.

In obigem Programmbeispiel wurden alle Konstanten mit EQU am Programmanfang definiert. In der Praxis definiere ich nur die wichtigsten Konstanten im Programmkopf mit EQU. Bei der Anweisung:

```
MOV AH,09
INT 21
```

ist für mich sofort ersichtlich, daß es sich hier um eine Textausgabe unter DOS handelt. Die im Beispiel verwendete Sequenz:

```
NUMOFF:     MOV AH,DOS_Txt       ; DOS-Display Code
            MOV DX,OFFSET TEXT ; Textadresse
            INT 21               ; DOS-Ausgabe
```

ist nach meiner Ansicht nicht so transparent. Letztlich ist es aber Geschmacksache, welche Konstanten direkt im Programm eingesetzt werden und welche global im Programmkopf definiert werden. Zusätzlich lassen sich auch Ausdrücke mit EQU definieren. Die Anweisungen:

```
Len   EQU 10T
Step  EQU  3
Lang  EQU Len * Step
```

weist der Konstanten Lang den Wert 30 (dezimal) zu.

Weiterhin lassen sich die EQU-Definition verwenden um einen Interrupt mit einem symbolischen Namen zu versehen. Der INT 3 wird häufig von Debuggern zum Test verwendet. Die CPU führt im Single-Step-Mode nach jedem Befehl einen INT 3 aus. Es besteht daher die Möglichkeit, den INT 3 durch:

```
TRAP EQU INT 3
```

umzudefinieren. Immer wenn im Programm der Begriff TRAP auftritt, ersetzt der Assembler diesen durch die INT 3-Anweisung.

Mit EQU-Anweisungen kann man in den meisten Assemblern einem Namen durchaus mehrfach verschiedene Werte innerhalb des Programmes zuweisen. Im MASM sind Mehrfachdeklarationen mit EQU unzulässig. Mit dem Operator = kann diese Restriktion allerdings umgangen werden. Er ist als Synonym für EQU zu verwenden. Dann können einem Namen im Verlauf des Programms durchaus unterschiedliche Werte zugewiesen werden.

Das $-Symbol

Mit diesem Symbol wird im Assembler die aktuelle Adresse symbolisiert. Um zum Beispiel die Länge einer Stringdefinition zu bestimmen, ist folgende Konstruktion erlaubt:

```
Text DB 'Hallo dies ist ein String'
Len EQU ($ - Text)
```

Der Assembler weist dann der Konstanten *Len* die Länge des Strings *Text zu.*

6.9 4.10 Operationen auf Ausdrücken

Bei der Erstellung von Assemblerprogrammen werden häufig Konstante in Ausdrücken verwendet (z.B: AND AX,3FFF). Die Konstante ist dabei in der geeigneten Form im Programm anzugeben. Eine Möglichkeit besteht darin, als Programmierer den Wert der Konstanten zu berechnen und im Quellprogramm einzusetzen. Dies ist aber nicht immer erwünscht: so kann es durchaus fehlerträchtig sein, wenn mehrere Werte manuell addiert werden. Um das Programm möglichst transparent zu gestalten, möchte man häufig die ursprünglichen Teilwerte im Programm mit angeben. Die meisten Assembler unterstützen diese Form und berechnen zur Übersetzungzeit den Wert eines Ausdruckes. Der TASM bietet eine Reihe solcher Operatoren, die nachfolgend kurz aufgeführt werden.

6.9.1 Addition

Der Operator erlaubt die Addition mehrere Konstante innerhalb eines Ausdruckes. Die Addition kann dabei durch ein + Zeichen markiert werden.

```
K22 EQU 20 + 02
Muster EQU Muster1 + Muster2
MOV AX,04C00 + 22  ; Berechne Konstante
```

Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen, sowie Kommazahlen verwendet werden.

6.9.2 Subtraktion

Der Operator erlaubt die Subtraktion von Konstanten innerhalb eines Ausdruckes. Die Subtraktion wird dabei durch das - Zeichen markiert (z.B. MOV AX,033-030). Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen, sowie Kommazahlen verwendet werden. Bei Variablen muß der Typ der beiden Operatoren übereinstimmen.

6.9.3 Multiplikation und Division

Die Operatoren erlaubte die Multiplikation und Division von Konstanten innerhalb eines Ausdruckes. Die Operatoren dürfen nur mit Kommazahlen oder auf ganzen Zahlen durchgeführt werden.

```
CMP CL, 2 * 3          ; Compare mit 6
MOV DX, 256 / 16       ; lade DX mit 16
MOV BX, 4 MOD 2        ; lade BX mit 0
```

Mit Hilfe dieser Operatoren lassen sich Berechnungen zur Assemblierungszeit ausführen. Dies erspart die manuelle (und fehleranfällig) Berechnung durch den Programmierer. Weiterhin wird im Listing sofort sichtbar wie die Konstante berechnet wird.

6.9.4 Schiebeoperatoren (SHL, SHR)

Mit den Operatoren SHR und SHL lassen sich Schiebeoperationen auf einer Konstanten oder einem Ausdruck ausführen. Die Operatoren besitzen das Format:

```
Operand SHR Count  (shift right)
Operand SHL Count  (shift left)
```

Die Schiebebefehle erlauben es, den Operanden bitweise nach links oder rechts zu verschieben. Der zweite Operand *Count* gibt dabei die Zahl der zu verschiebenden Binärstellen an. Die Bits, die in den Operanden eingeschoben werden, sind mit dem Wert 0 belegt. Beispiel für die Anwendung der Operatoren sind:

```
MOV BX,0FF33 SHR 4          ; BX = 0FF3
MOV DX,01 SHL 4             ; DX = 010
```

6.9.5 Logische Operatoren

Mit den Anweisungen AND, OR, XOR und NOT lassen sich logische Operationen auf den Operanden ausführen. Die Operatoren besitzen das Format:

```
Operand AND Operand
Operand OR Operand
Operand XOR Operand
NOT Operand
```

Die Befehle dürfen ausschließlich mit vorzeichenlosen Byte- oder Word-Konstanten benutzt werden. Die Anweisung:

```
MOV AL,03F AND 0F
```

blendet die oberen 4 Bits der Konstanten 3FH aus. Für die Verknüpfung der Operanden gelten die Regeln für AND, OR, XOR und NOT. Der NOT-Operator invertiert den Wert der angegebenen Konstanten.

6.10 Vergleichsoperatoren

Der TASM bietet einen weiteren Set an Vergleichsoperatoren:

Operand EQ Operand (equal)
Operand NE Operand (not equal)
Operand LT Operand (less than)
Operand LE Operand (less or equal)
Operand GT Operand (greater then)
Operand GE Operand (greater then or equal)

Als Operanden müssen vorzeichenlose Ganzzahlen (Byte oder Word) angegeben werden. Dabei können sowohl Konstante als auch Variable benutzt werden (z.B. MASKE1 EQ Mode). Als Ergebnis wird ein Byte oder Word zurückgegeben, welches die Werte *true* (0FFFFH) oder *false* (0) enthält. Die Anweisung:

```
MOV AL, 4 EQ 3
```

lädt AL mit dem Wert 0, da der Ausdruck falsch ist.

Die bisher besprochenen Operanden lassen sich in Ausdrücken kombinieren. Dabei gelten die folgenden Prioritäten:

1. Klammern
2. Punkt
3. Segment Override, PTR
4. OFFSET, SEG, TYPE
4. HIGH, LOW
5. +, -
6. *, /, MOD, SHR, SHL
7. +, - (binär)
8. EQ, NE, LT, LE, GT, GE
9. NOT
10. AND
11. OR, XOR
12. SHORT

bei der Auswertung eines Ausdruckes. Die Zahl 1 definiert dabei die höchste Priorität.

6.11 Die EVEN Directive

Mit dieser Anweisung erzwingt der Programmierer, daß der nächste assemblierte Befehl oder die Lage einer Variablen auf einer geraden Speicheradresse festgelegt

wird. Innerhalb des Codesegmentes fügt der TASM gegebenenfalls eine NOP-Anweisung ein. Bei Variablen werden diese einfach auf eine gerade Adresse gelegt. Die EVEN-Directive ist besonders bei 80X86-Prozessoren hilfreich, da Zugriffe auf gerade Adressen schneller ausgeführt werden. Ein 16-Bit-Zugriff auf eine ungerade Adresse erfordert 2 Speicheroperationen.

Programmbeispiel

Doch nun möchte ich ein weiteres Programmbeispiel vorstellen. Erstmalig soll nun eine EXE-Datei erstellt werden. Das Programm besitzt den Namen:

ASK.EXE

und erlaubt Benutzerabfragen aus Batchdateien. Das Programm wird mit:

ASK <Text>

aufgerufen. Der Text innerhalb der Aufrufzeile ist dabei optional. Nach dem Start wird der Text der Aufrufzeile am Bildschirm ausgegeben. Dann wird die Tastatur abgefragt und das Zeichen der gedrückten Taste wird an DOS zurückgegeben. Der Tastencode läßt sich innerhalb der Batchdatei dann durch die ERRORLEVEL-Funktion abfragen.

Mit dem Aufruf:

ASK Abbruch (J/N) ?

gibt ASK die Nachricht:

Abbruch (J/N)?

aus und wartet auf eine Benutzereingabe. ASK terminiert sofort nach der Eingabe. Der Code des Zeichens läßt sich in DOS dann per ERRORLEVEL abfragen:

```
ASK Abbruch (J/N)?
IF ERRORLEVEL 75 GOTO NO
IF ERRORLEVEL 74 GOTO YES
:NO .....
```

Zu beachten ist lediglich, daß ERRORLEVEL die Codes zwischen 0 und 255 auf >= abprüft. Beispiele für den Einsatz des Programmes ASK finden sich im Anhang und in /2/. Doch nun möchte ich das dazugehörige Listing vorstellen.

```
;==================================
; File: ASK.ASM   (c) Born G.
; Version: V 1.0  TASM
; Funktion: Programm zur Benutzerab-
```

```
; frage in Batchdateien. Aufruf:
;
;     ASK <Text>
;
; Der Text wird auf dem Screen ausge-
; gegeben. Der Tastencode wird an DOS
; zurückgegeben. (Bei Funktionstasten
; wird FF zurückgegeben). Er läßt sich
; per ERRORLEVEL abfragen. Das Programm
; ist als COM-Datei zu übersetzen!!
;==================================
;
       .MODEL SMALL
       .RADIX 16            ; Hexadezimalsystem

       Blank EQU 20         ; Blank
       Err1  EQU 0FF        ; Error

       .STACK               ; 1 K Stack

       .DATA
;
; Bereich mit den Textkonstanten
;
Crlf  DB 0D, 0A,"$"
;

       .CODE
;
ASK:   CALL NEAR PTR Text   ; Textausgabe
       CALL NEAR PTR Key    ; Abfrage der Tastatur
       PUSH AX              ; merke Tastencode
;
; CR,LF ausgeben
;
       MOV AX,SEG Crlf      ; lese Segment Text
       PUSH DS              ; merke DS-Inhalt
       MOV DS,AX            ; setze Segmentadr.
       MOV AH,09            ; INT 21-Stingausgabe
       MOV DX,OFFSET Crlf   ; Stringadresse
       INT 21               ; ausgeben
       POP DS               ; restauriere DS
       POP AX               ; restauriere Tastencode
;
; DOS-Exit, Returncode steht bereits in AL
;
       MOV AH,4C            ; INT 21-Exitcode
       INT 21               ; terminiere
;

Text   PROC NEAR
;--------------------------------
; Unterprogramm zur Ausgabe des
; Textes aus der Kommandozeile
;--------------------------------
; ermittle Lage des PSP über undok.
; Funktion 51 des INT 21
;
       MOV AH,51            ; DOS-Code
```

```
       INT 21              ; Get PSP
       MOV ES,BX           ; ES:= PSP-Adr
;
; prüfe ob Text im PSP vorhanden ist
;
       MOV CL,ES:[80]      ; lese Pufferlänge
       CMP CL,0            ; Text vorhanden ?
       JZ  Ready           ; Nein -> Exit
;
; Text ist vorhanden, ausgeben per INT 21, AH = 02
;
       MOV BX,0082         ; Zeiger auf 2. Zeichen
       DEC CL              ; 1 Zeichen weniger
Loop1:           ; Beginn der Ausgabeschleife !!!!
       MOV AH,02           ; INT 21-Code Display Char.
       MOV DL,ES:[BX]      ; Zeichen in DL laden
       INT 21              ; CALL DOS-Ausgabe
       INC BX              ; Zeiger nächstes Zchn
       DEC CL              ; Zeichenzahl - 1
       JNZ Loop1           ; Ende ? Nein-> Loop
;
       MOV AH,02           ; INT 21-Code Display
       MOV DL,Blank        ; Blank anhängen
       INT 21              ; und ausgeben
Ready: RET                 ; Ende Unterprogramm
Text   ENDP

;
Key    PROC NEAR
;--------------------------------
; Unterprogramm zur Tastaturabfrage
; benutze INT 21, AH = 08 Read Keyboard
; oder:           AH = 01 Read Keyboard & Echo
;--------------------------------
; lese 1. Zeichen
       MOV AH,01           ; INT 21-Read Key & Echo
       INT 21              ; Read Code
       CMP AL,0            ; Extended ASCII-Code ?
       JNZ Exit            ; Nein -> Ready
;
; lese 2. Zeichen beim Extended ASCII-Code
;
       MOV AH,08           ; INT 21 Read Keyboard
       INT 21              ; Code aus Puffer lesen
       MOV AL,Err1         ; Fehlercode setzen
Exit:  RET                 ; Ende Unterprogramm
Key    ENDP

END Ask
```

Listing 6.4: ASK.ASM

Das Programm benutzt eine Reihe neuer Funktionen, auf die ich kurz eingehen möchte. Zuerst einmal die Technologie der DOS-Aufrufe zur Erledigung der Aufgabe. Zur Ausgabe der Benutzermeldung ist die INT 21-Funktion AH = 02 zu nutzen, die ein Zeichen im Register DL erwartet und dieses Zeichen ausgibt. Dann fragt ASK die

Tastatur ab. Mit der INT 21-Funktion AH = 01H wartet DOS solange, bis ein Zeichen eingegeben wird. Das Zeichen wird bei der Eingabe durch DOS auf den Ausgabebildschirm kopiert. So sieht der Benutzer seine Eingabe. Nun besitzt ein DOS-Rechner Tasten, die nicht ein Byte, sondern zwei Byte (Extended ASCII Code) zurückgeben. Funktionstasten gehören zu dieser Kategorie. Wird eine Taste mit erweitertem Code (Extended ASCII-Code) betätigt, hat das erste gelesene Byte den Wert 00H. Dann muß das Programm den Tastaturpuffer ein weiteres mal lesen um das zweite Zeichen zu entfernen. Hierzu wird die INT 21-Funktion 08H benutzt, die kein Echo der eingegebenen Zeichen zuläßt. Um innerhalb eines Batchprogramms zu erkennen, daß eine Funktionstaste gedrückt wurde, sollte ASK in diesem Fall den Wert FFH an DOS zurückgeben. Damit entfallen die Funktionstasten zur Eingabe. Die INT 21-Funktionen AH = 01H und AH = 08H geben das gelesene Zeichen im Register AL zurück. Wird nun die INT 21-Funktion AH = 4CH aufgerufen, terminiert das Programm ASK. Der Code im Register AL läßt sich dann über ERRORLEVEL abfragen. Mit diesem Wissen sollte sich das Problem eigentlich lösen lassen.

Leider bringt die Verwendung von EXE-Dateien eine Reihe weiterer Schwierigkeiten mit sich. So kann die Lage des PSP nicht mehr so einfach wie bei COM-Dateien ermittelt werden. Vielmehr ist der (undokumentierte) INT 21-Aufruf AH=51H zu nutzen. Dieser gibt im Register BX die Lage des PSP-Segmentes zurück.

6.12 Die PROC-Directive

Um das Programm ASK transparenter zu gestalten wurden einzelne Funktionen in Unterprogramme verlagert. Die Ausgabe des Textes der Kommandozeile erfolgt in dem Unterprogramm *Text*. Für die Tastaturabfrage ist das Modul *Key* zuständig. Die Prozeduren werden mit den Schlüsselworten:

```
xxx   PROC NEAR
  ....
  xxxx  ENDP
```

eingeschlossen. Damit erkennt der Assembler, daß er ein RET-NEAR als Befehl einsetzen muß. Beim Aufruf der Prozedur kann TASM auch prüfen, ob der Abstand größer als 64 KByte wird.

Diese Anweisung muß bei TASM am Beginn eines Unterprogrammes (Prozedur) stehen. Dabei sind die Schlüsselworte:

```
Name PROC FAR
Name PROC NEAR
Name PROC
```

zulässig.

Name steht dabei für den Namen der Prozedur, während die Schlüsselworte FAR und NEAR die Aufrufform für den CALL-Befehl festlegen. Sobald ein RET-Befehl auftritt, ersetzt der Assembler diesen durch die betreffende RET- oder RETF-Anweisung. Der Programmierer braucht also nur einen Befehlstyp für RET einzusetzen und der Assembler ergänzt den Befehl. Viele Programmierer ziehen aber vor, den RET-Befehl explizit (notfalls auch als RETF) zu schreiben. Dies ist bei der Wartung älterer Programme hilfreich, da sofort erkennbar wird, ob eine Prozedur mit FAR oder NEAR aufzurufen ist. Wird eine Prozedur mit PROC definiert, muß am Ende des Programmes das Schlüsselwort ENDP stehen. Da das SMALL-Model vereinbart wurde, sind alle Unterprogrammaufrufe als NEAR kodiert.

6.12.1 Anmerkungen

Bei einem EXE-Programm läßt sich ein Datensegment mit der DATA-Directive erzeugen. Ein Problem ist die Zuweisung eines korrekten Wertes für das Segmentregister DS. Der TASM bietet hierzu die folgende Möglichkeit:

```
MOV AX,@DATA
MOV DS,AX
```

die die erforderlichen Anweisungen erzeugen.

Um aus dem Quellprogramm eine EXE-Datei zu erzeugen, sind mehrere Schritte erforderlich:

- Erstellen der Quelldatei
- Übersetzen in einen OBJ-FIle
- Linken des OBJ-Files zu einer EXE-Datei

Die Übersetzung der Quelldatei in eine OBJ-Datei erfolgt mit der Anweisung:

```
TASM ASK.ASM,,,,
```

Der Assembler legt seine Fehlermeldungen dann in der Listdatei ASK.LST ab. Bei einer fehlerfreien Übersetzung ist die OBJ-Datei in eine EXE-Datei zu linken:

```
LINK ASK.OBJ
```

Der Linker wird dann die Namen der EXE-, MAP- und LIB-Dateien abfragen. Um diese Abfrage zu umgehen, kann hinter dem OBJ-Namen eine Serie von Kommas folgen:

```
LINK ASK.OBJ,,,,
```

Damit wird LINK signalisiert, daß die Namen der jeweiligen Ausgabedateien aus dem Namen der Eingabedatei (hier ASK) zu extrahieren ist.

Sobald die Datei ASK.EXE vorliegt, läßt sie sich mit einem Debugger testen.

6.13 Erstellen einer EXE-Datei aus mehren OBJ-Files

Nun möchte ich noch einen Schritt weitergehen und ein EXE-Programm aus mehreren Modulen aufbauen. Bei größeren Programmen möchte man nicht alle Unterprogramme in einer Quelldatei stehen haben. Vielmehr wird man einzelne Unterprogramme und Module in getrennten Dateien halten. Dies erleichtert das Editieren und Übersetzen. Sobald die Datei in einen OBJ-File übersetzt wurde, läßt sich dieser mit LINK zu den Hauptprogrammen zubinden. Diese Technik möchte ich kurz vorstellen.

Programmbeispiel

Unser Beispielprogramm erhält den Namen:

ESC.EXE

und erlaubt die Ausgabe von Zeichen an die Standard-Ausgabeeinheit. Die Zeichen lassen sich als Strings oder Hexzahlen beim Aufruf eingeben. Damit gilt folgende Aufrufsyntax:

ESC <Param1> <Param2> .... <Param n>

Die Parameter <Param x> enthalten die auszugebenden Zeichen als:

- Hexzahl
- String

Die Parameter sind durch Kommas oder Leerzeichen zu trennen. Ein String zeichnet sich dadurch aus, daß die Zeichen durch Anführungszeichen eingerahmt werden. Mit ESC lassen sich zum Beispiel sehr komfortabel Steuerzeichen an den Drucker übergeben. Folgende Zeilen enthalten Beispiele für den Aufruf des Programmes. Die Kommentare sind nicht Bestandteil des Aufrufes:

```
ESC 41 42 43          ; erzeuge ABC auf Screen
ESC 0C > PRN:         ; Seitenvorschub auf PRN:
ESC 1B 24 30 > PRN:   ; Steuerzeichen an PRN:
```

```
ESC "Hallo" 0A 0D       ; Hallo auf Screen
ESC "Text" > A.BAT      ; in Datei schreiben
```

In der Vergangenheit hat mir das Programm gute Dienste geleistet. So läßt sich mit einigen Batchbefehlen der Drucker auf beliebige Schriftarten einstellen. Im Anhang ist ein Batchprogramm als Beispiel enthalten. Weitere Hinweise finden sich in /2/.

Das Programm ESC.EXE wird in drei Quelldateien aufgeteilt. Eine Datei enthält ein Unterprogramm zur Wandlung einer Hexzahl in Form eines ASCII-Strings in den entsprechenden Hexadezimalwert. Die Quelldatei besitzt folgendes Format:

```
;=========================================
; File : HEXASC.ASM  (c) G. Born
; Version: 1.0  (TASM)
; Convert ASCII-> HEX
; CALL: ES:DI -> Adresse 1.Ziffer (XX)
;       CX       Länge Parameterstring
; Ret.: CY : 0  o.k.
;       AL      Ergebnis
;       ES:DI -> Adresse nächstes Zeichen
;       CY : 1  Fehler
;=========================================
;
       .MODEL SMALL
       .RADIX 16           ; Hexadezimalsystem
       PUBLIC HexAsc       ; Label global

Blank  EQU 20              ; Leerzeichen
Null   EQU 30

       .CODE
;
HexAsc: MOV AL,ES:[DI]     ; lese ASCII-Ziffer
        CMP AL,'a'         ; Ziffer a - f ?
        JB L1              ; keine Kleinbuchstaben
        SUB AL,Blank       ; in Großbuchstaben
L1:     CMP AL,Null        ; Ziffer 0 - F ?
        JB Error1          ; keine Ziffer -> Error
        CMP AL,'F'         ; Ziffer 0 - F ?
        JA Error1          ; keine Ziffer -> Error
        SUB AL,Null        ; in Hexzahl wandeln
        CMP AL,9           ; Ziffer > 9 ?
        JBE Ok             ; JMP OK
        SUB AL,07          ; korr. Ziffern A..F
        JO  Error1         ; keine Ziffer -> Error
Ok:     CLC                ; Clear Carry
        RET                ; Exit
; -> setze Carry
Error1: STC                ; Error-Flag
        RET                ; Exit
;
        END HexAsc
```

Listing 6.5: HEXASC.ASM

Das Unterprogramm erwartet in ES:DI einen Zeiger auf das erste Zeichen der zu konvertierenden Zahl. HEXASC konvertiert immer zwei Ziffern zu einem Byte. Das Ergebnis wird in AL (als Byte) zurückgegeben. Ist das Carry-Flag gelöscht, dann ist der Wert gültig. Bei gesetztem Carry-Flag wurde eine ungültige Ziffer gefunden und das Ergebnis in AL ist undefiniert.

In einer zweiten Quelldatei wurden zusätzliche Hilfsmodule untergebracht:

SKIP

Aufgabe dieses Moduls ist es, innerhalb des Eingabestrings die Separatorzeichen Blank und Komma zu erkennen. Der Eingabestring wird durch ES:DI adressiert. Erkennt SKIP einen Separator, wird der Lesezeiger solange erhöht, bis kein Separatorzeichen mehr vorliegt oder das Ende des Strings erreicht wurde. Wurde das Ende des Strings erreicht, markiert SKIP dies durch ein gesetztes Carry-Flag. Der Lesezeiger ES:DI zeigt nach dem Aufruf immer auf das nächste Zeichen.

String

Kommt in der Eingabezeile Text vor:

ESC "Hallo" 0D 0A

wird *String* benutzt um den Text direkt an die Ausgabeeinheit auszugeben. Die Funktion erwartet den Lesezeiger auf den Text in ES:DI. Wird im ersten Zeichen ein " gefunden, gibt das Unterprogramm die folgenden Zeichen aus. Das Unterprogramm terminiert, sobald ein zweites Anführungszeichen auftritt, ohne daß das String-Ende erreicht ist. Wird das Ende der Parameterliste erreicht, ohne daß der String beendet wurde (zweites Anführungszeichen " fehlt), gibt das Modul ein gesetztes Carry-Flag. Nach der Ausgabe des Strings zeigt der Lesezeiger auf das nächste Zeichen hinter dem String.

Number

Dieses Modul wertet die Parameterzeile aus und liest eine Hexadezimalzahl mit 2 Ziffern ein. Anschließend wird das Unterprogramm HEXASC zur Konvertierung aufgerufen. Der zurückgegebene Wert wird dann an die Ausgabeeinheit gesendet. Das Unterprogramm erwartet wieder einen Lesezeiger in den Registern ES:DI. Nach dem Aufruf wird das Zeichen hinter der Hexzahl durch diesen Zeiger adressiert. Ist das Carry-Flag gesetzt, wurde das Ende der Parameterzeile erreicht. Falls ein fehlerhaftes Zeichen erkannt wird, terminiert das Programm über die DOS-Exit-Funktion.

Das Quellprogramm besitzt folgenden Aufbau:

```
;=========================================
; File : MODULE.ASM  (c) G. Born
; Version: 1.0 (TASM)
; File mit den Modulen zur Ausgabe und Be-
; arbeitung der Zeichen.
;=========================================
;
       .MODEL SMALL
       .RADIX 16           ; Hexadezimalsystem
       EXTRN HexAsc:NEAR   ; Externes Modul

       .CODE
;
;=========================================
; SKIP Separator (Blank, Komma)
;
; Aufgabe: Suche die Separatoren Blank oder
;          Komma und überlese sie.
;
; CALL: ES:DI -> Adresse ParameterString
;       CX       Länge Parameterstring
; Ret.: CY : 0  o.k.
;       ES:DI -> Adresse 1. Zchn. Parameter
;       CY : 1  Ende Parameterliste erreicht
;=========================================
;
       PUBLIC Skip
;
Skip   PROC NEAR
Loops:CMP CX,0000     ; Ende Parameterliste ?
      JNZ Test1       ; Nein -> JMP Test
      STC             ; markiere Ende mit Carry
      JMP SHORT Exit2 ; JMP Exit2
Test1:CMP BYTE PTR ES:[DI],' ' ; Zchn. = Blank ?
      JZ  Skip1       ; Ja -> Skip
      CMP BYTE PTR ES:[DI],',' ; Zchn. = "," ?
      JNZ Exit1       ; Nein -> Exit1
Skip1:DEC CX          ; Count - 1
      INC DI          ; Ptr to next Char.
      JMP NEAR PTR Loops  ; JMP Loop
Exit1:CLC             ; Clear Carry
Exit2:RET
Skip  ENDP
;
;-----------------------------------------
; Display String
;
; Aufgabe: Gebe einen String innerhalb der
;          Parameterliste aus.
;
; CALL: ES:DI -> Adresse "......" String
;       CX       Länge Parameterstring
; Ret.: CY : 0  o.k.
;       ES:DI -> Adresse nach String
;       CY : 1  Ende Parameterliste erreicht
;-----------------------------------------
;
       PUBLIC String
;
```

```
String PROC NEAR

Begin: CMP BYTE PTR ES:[DI],22 ; " gefunden ?
       JNZ Exit3       ; kein String -> EXIT
       INC DI          ; auf nächstes Zeichen
       DEC CX          ; Count - 1
Loop1: CMP CX,0000     ; Ende Parameterliste ?
       JNZ Test2       ; Nein -> JMP Test
       STC             ; markiere Ende mit Carry
       RET             ; Exit
Test2: CMP BYTE PTR ES:[DI],22 ; " -> Stringende ?
       JZ  Ende        ; Ja -> JMP Ende
Write: MOV DL,ES:[DI]  ; lese Zeichen
       MOV AH,02       ; DOS-Code
       INT 21          ; ausgeben
       DEC CX          ; Count - 1
       INC DI          ; Ptr to next Char.
       JMP NEAR PTR Loop1  ; JMP Loop
Ende:  INC DI          ; auf Stringende
       DEC CX          ; Count - 1
Exit3: CLC             ; ok-> clear Carry
       RET
String ENDP
;
;-----------------------------------------
; Display Number
;
; Aufgabe: Gebe eine HEX-Zahl innerhalb der
;          Parameterliste aus.
;
; CALL: ES:DI -> Adresse 1.Ziffer (XX)
;       CX       Länge Parameterstring
; Ret.: CY : 0  o.k.
;       ES:DI -> Adresse nach Zahl
;       CY : 1  Ende Parameterliste erreicht
;-----------------------------------------
;
       PUBLIC Number
;
Number  PROC NEAR

        CALL NEAR PTR HexAsc ; 1. Ziffer konvert.
        JC Error         ; keine Ziffer -> Error
        MOV DL,AL        ; merke Ergebnis
; 2. Ziffer lesen
        INC DI           ; nächstes Zeichen
        DEC CX           ; Zähler - 1
        CMP CX,0000      ; Pufferende ?
        JZ Displ1        ; JMP Display
        CALL NEAR PTR HexAsc  ; 2. Ziffer konvert.
        JC Displ1        ; no Ziffer ->JMP Displ1
;
; schiebe 1. Ziffer in High Nibble
;
        PUSH CX          ; schiebe 1. Ziffer
        MOV CL,04        ; in High Nibble
        SHL DL,CL
        POP CX
        OR  DL,AL        ; Low Nibble einblenden
```

```
Displ1: MOV AH,02         ; DOS-Code
        INT 21            ; Code in DL ausgeben
        AND CX,CX         ; Ende Parameterliste?
        STC               ; set Carry zur Vorsicht
        JZ Exit4          ; Ende ereicht -> Exit
        INC DI            ; auf nächstes Zeichen
        DEC CX            ; Count - 1
        CLC               ; Clear Carry
Exit4:  RET
;-----------------------------------------
; Ausgabe des Fehlertextes und Exit zu DOS
;-----------------------------------------
Error: MOV AX,SEG Txt    ; DS: auf Text
       MOV DS,AX
       MOV AH,09         ; DOS-Code
       MOV DX,OFFSET Txt ; Adr. String
       INT 21            ; Ausgabe
       MOV AX,4C01       ; DOS-Code
       INT 21            ; Exit
;
;=========================================
; Fehlertext
;=========================================
Txt DB "Fehler in der Parameterliste",0D,0A,"$"
;
Number ENDP
       END
```

Listing 6.6: MODULE.ASM

Damit wird das eigentliche Hauptprogramm sehr kurz. Es muß lediglich die Adresse der DOS-Kommandozeile ermitteln und dann die Parameter auswerten. Zur Auswertung werden dann die Unterprogramme benutzt. Das Programm besitzt folgenden Aufbau:

```
;=========================================
; File : ESC.ASM  (c) G. Born
; Version: 1.0 (TASM)
; Programm zur Ausgabe von Zeichen an die
; Standardausgabeeinheit. Das Programm wird
; z.B. mit: ESC 0D,0A,1B "Hallo"
; von der DOS-Kommandoebene aufgerufen und
; gibt die spezifizierten Codes aus.
;=========================================
;
       .MODEL SMALL
       .RADIX 16           ; Hexadezimalsystem

       Blank EQU 20        ; Blank

       EXTERN HexAsc:NEAR  ; externe Module
       EXTERN String:NEAR
       EXTERN Number:NEAR
       EXTERN Skip:NEAR

;=========================================
```

```
; Definition des Stacksegmentes
;=========================================
       .STACK 200H

       .CODE
;
;=========================================
; Hauptprogramm
;=========================================
;
Start: MOV AH,51       ; ermittle Adr. PSP
       INT 21          ; "
       MOV ES,BX       ; ES = Adr. PSP !
       MOV CL,ES:[80]  ; lade Pufferlänge
       CMP CL,0        ; Text vorhanden
       JZ  Endx        ; kein Text vorhanden
       XOR CH,CH       ; clear Counter Hbyte
       MOV DI,0081     ; lade Puffer Offset
Loopb:                 ; gebe Parameter aus
       AND CX,CX       ; Ende erreicht ?
       JZ Endx         ; DOS-Exit
       CALL Skip       ; Separatoren überlesn
       JC Endx         ; DOS-Exit
       CALL String     ; String ausgeben
       JC Endx         ; DOS-Exit
       CALL Skip       ; Separ. überlesen
       JC Endx         ; DOS-Exit
       CALL Number     ; Zahl ausgeben
       JC Endx         ; DOS-Exit
       JMP Loopb       ; next Param.
;
; DOS-Exit, Returncode in AL
;
Endx:  MOV AX,4C00     ; DOS Exitcode
       INT 21
;
       END Start
```

Listing 6.7: ESC.ASM

Die einzelnen Quelldateien sind mit den Anweisungen:

TASM ESC.ASM,,,,
TASM MODULE.ASM,,,,
TASM HEXASC.ASM,,,,

zu übersetzen. Der Macroassembler wird zwar einige Fehlermeldungen bringen. So signalisiert er zum Beispiel, daß in ASK verschiedene Prozeduren (Skip, etc.) aufgerufen werden, die nicht in der Quelldatei vorhanden sind. Dies ist nicht weiter tragisch, daß anschließend die einzelnen OBJ-Files mit LINK kombiniert werden:

TLINK ESC+MODULE+HEXASC,,,,,

Damit werden die OBJ-Files zu einem lauffähigen EXE-Programm kombiniert. Der Linker löst auch die vom Assembler gemeldeten externen Referenzen auf. Sofern der Linker keine Fehlermeldungen mehr bringt kann nun das Programm ESC.EXE mit einem Debugger getestet werden.

6.14 Die Directiven für externe Module

In obigen Quellprogramme finden sich einige neue Anweisungen für den Assembler die kurz vorstellen möchte. Die nachfolgenden Directiven beziehen sich auf getrennt zu assemblierende Module und sind für den Linker erforderlich.

6.14.1 Die PUBLIC-Directive

Diese Anweisung erlaubt die explizite Auflistung von Symbolen (Variable, Prozeduren, etc.), die durch andere Module adressierbar sein sollen. Der Linker wertet diese Informationen aus den .OBJ-Dateien aus und verknüpft offene Verweise auf diese Symbole mit den entsprechenden Adressen. So kann zu einem Programm eine Prozedur aus einer fremden Objektdatei zugebunden werden. Der Linker setzt dann im CALL-Aufruf die Adresse des Unterprogrammes ein. Weiterhin lassen sich mit der PUBLIC-Directive Variable als global definieren, so daß andere Module darauf zugreifen können. Die Anweisung sollte im Modulkopf stehen und kann folgende Form besitzen:

PUBLIC Skip, Number, String

Die Definitionen dürfen aber auch direkt vor dem jeweiligen Unterprogramm angegeben werden.

Die Symbole Skip, String und Number lassen sich dann aus anderen Modulen (hier aus ESC) ansprechen, wenn sie dort als EXTRN erklärt werden. Fehlt die PUBLIC-Anweisung, wird der Linker eine Fehlermeldung mit den offenen Referenzen angeben. Zur Erhöhung der Programmtransparenz und zur Vermeidung von Seiteneffekten sollten nur die globalen Symbole in der PUBLIC-Anweisungsliste aufgeführt werden.

6.14.2 Die EXTRN-Directive

Diese Directive ist das Gegenstück zur PUBLIC-Directive. Soll in einem Programm auf ein Unterprogramm oder eine Variable aus einem anderen .OBJ-File zugegriffen werden, muß dies dem TASM bekannt sein. Mit der EXTRN-Directive nimmt der TASM an, daß das Symbol in einer externen .OBJ-Datei abgelegt wurde und durch den Linker überprüft wird. Die EXTRN-Directive besitzt folgendes Format:

EXTRN Name1:Typ, Name2:Typ, ...

Als Name ist der entsprechende Symbolname einzutragen. Weiterhin muß der Typ der Referenz in der Deklaration angegeben werden. Hierfür gilt:

BYTE oder NEAR
WORD oder ABS
DWORD
QWORD
FAR

Das Gegenstück zu obiger PUBLIC-Definition ist z.B. die Erklärung der externen Referenzen im Hauptprogramm:

EXTRN Skip:NEAR, String:NEAR, Number:NEAR

Tritt nun eine Referenz auf ein solches Symbol auf (z.B: CALL NEAR PTR Skip), generiert der TASM einen Unterprogrammaufruf ohne die absolute Adresse einzutragen. Die Adresse wird beim LINK-Aufruf dann nachgetragen.

6.14.3 Die END-Directive

Diese Directive signalisiert das Ende des Assemblermoduls. Die Anweisung besitzt die Form:

END
END start_adr

wobei der Name *start_adr* als Label für Referenzen dienen darf. TASM behandelt den hinter END angegebenen Namen wie ein Symbol, welches mit EQU definiert wurde. So läßt sich dann die Programmlänge leicht ermitteln. Folgen weitere Anweisungen, werden diese als getrenntes Programm übersetzt. Damit lassen sich in einer Quellcodedatei mehrere Module ablegen und mit einem Durchlauf assemblieren.

6.14.4 Die GROUP-Directive

Mit dieser Directive wird der Linker angewiesen, alle angegebenen Programmsegmente innerhalb eines 64-KByte-Blocks zu kombinieren. Es gilt dabei die Syntax.

Group_name GROUP Seg_name1, Seg_name2, ...

Der Group_name läßt sich dann innerhalb der Module der Gruppe verwenden (z.B: MOV AX,Group_name). Passen die Module nicht in einen 64-KByte-Block, gibt der Linker eine Fehlermeldung aus. Mit dieser Anweisung lassen sich einzelne Module zu

Gruppen kombinieren und in einem Segment ablegen. Zu Beginn des Segmentes werden die Segmentregister gesetzt und sind für alle Module der Gruppe gültig. Wird die GROUP-Directive verwendet, muß sie allerdings in allen Modulen benutzt werden, da sonst einzelne Module nicht im Block kombiniert werden. Die GROUP-Anweisung wird in den Beispielprogrammen nicht benutzt.

6.15 Einbinden von Assemblerprogrammen in Hochsprachen

Abschließend möchte ich noch kurz auf die Einbindung eines Assemblerprogramms in Hochsprachen eingehen. Hierzu ist das Modul wie in obigem Beispiel als Prozedur in einer Quelldatei zu speichern und durch den TASM in eine OBJ-Datei zu übersetzen. Dann kann das Modul per LINK in eine Hochsprachenapplikation eingebunden werden.

Programmbeispiel

Nachfolgend möchte ich kurz an Hand eines kleinen Beispiels die Einbindung von Assemblerprogrammen in Turbo Pascal- und QuickBasic-Programme zeigen.

Aufgabe ist es eine Prüfsumme nach dem CRC16-Verfahren zu berechnen. Diese Operation wird bei der Datenübertragung zur Fehlerprüfung häufig verwendet. Die Realisierung einer CRC-Berechnung ist in Hochsprachen nicht effizient möglich. Deshalb wird die Routine in Assembler kodiert und als OBJ-File in die Hochsprachenapplikation eingebunden.

Das Programm CRC.ASM übernimmt die Berechnung der CRC-Summe. Die Parameter werden per Stack übergeben. Das Modul ist als OBJ-File zu übersetzen. Der genaue Aufbau der Parameterübergabe und des Programmes ist nachfolgendem Listing zu entnehmen.

```
;-------------------------------------------------
; File       : CRC.ASM
; Version    : 1.1 (TASM)
; Autor      : (c) G. Born
; Funktion   :
;
;        CRC 16 Generator für den 8086 Prozessor
;
; Es wird das Polynom y = x^16 + x^15 + x^2 + 1
; verwendet
; Aufruf von Hochsprachen mit:
;
; CALL CRC (CRCr,Buff,Len)
;
; CRCr = Adresse Variable 16 Bit CRC Register
```

```
; Buff = Adresse des Zeichenpuffers
; Len  = Wert Zeichenzahl im Puffer
;
; Es sind die Adressen der zwei Parameter CRCr,
; Buff und der Wert von Len über den Stack zu
; übergeben:
;
;
;
;       Stack Anordnung
;      +-------------------+
;      ! Seg:   CRCr       !
;      +-------------------+ + 0E
;      ! Ofs:   CRCr       !
;      +-------------------+ + 0C
;      ! Seg:   Buff       !
;      +-------------------+ + 0A
;      ! Ofs:   Buff       !
;      +-------------------+ + 08
;      ! Wert   Len        !
;      +-------------------+ + 06
;      ! Seg:   Return Adr.!
;      +-------------------+ + 04
;      ! Ofs:   Return Adr.!
;      +-------------------+ + 02
;      !  BP  von CRC      !
;      +-------------------+
;
; Die Procedur ist als FAR aufzurufen. Es wird
; angenommen, daß die Variablen im DS-Segment
; liegen. Diese Konvention entspricht der
; Parameterübergabe in Turbo Pascal und Quick
; Basic. Für die eigentliche CRC-Berechnung
; gilt folgende Registerbelegung:
;
; Register: BX = CRC Register
;           SI = Zeiger in den Datenstrom
;           CX = Zahl der Daten
;           AH = Bit Counter
;           AL = Hilfsaccu
;
;-------------------------------------------------
;
        .RADIX    16
        .MODEL    LARGE

POLY     EQU       0010000000000001B ; Polynom

        .CODE

        PUBLIC    CRC
CRC     PROC      FAR

CRCX:   PUSH      BP            ; save old frame
        MOV       BP,SP         ; set new frame
        PUSH      DS            ; save DS
        MOV       CX,[BP]+06    ; get (Len)
        MOV       SI,[BP]+08    ; get Adr (Buff)
        MOV       DS,[BP]+0A    ; get Seg (Buff)
```

```
        MOV       BX,[BP]+0C    ; get Adr (CRC)
        MOV       DI,BX         ; merke Adr
        MOV       BX,DS:[BX]    ; load CRC value
;
        CLD                     ; Autoincrement
;
LESE:   MOV       AH,08         ; 8 Bit / Zeichen
        LODSB                   ; get Zchn & SI+1
        MOV       DL, AL        ; merke Zeichen
;
; CRC Generator
;
CRC1:   XOR       AL,BL         ; BCC-LSB XOR Zchn
        RCR       AL,1          ; Ergeb. Carry Bit
        JNC       NULL          ; Serial Quot.?
;
; Serial Quotient = 1
;
EINS:   RCR       BX,1          ; SHIFT CRC right
        XOR       BX,POLY       ; Übertrag einbl.
        JMP       TESTE         ; weitere Bits
;
; Serial Quotient = 0
;
NULL:   RCR       BX,1          ; Shift CRC right
;
TESTE:  MOV       AL,DL         ; Lese Zeichen
        RCR       AL,1          ; Zchn 1 Bit right
        MOV       DL,AL         ; merke Rest zchn
        DEC       AH            ; 8 Bit fertig ?
        JNZ       CRC1          ; Zchn ready ?
        LOOP      LESE          ; String ready?
;
; Ende -> Das CRC Ergebnis steht im Register BX
;      -> schreibe in Ergebnisvariable zurück
;
        MOV       DS:[DI],BX    ; restore CRC

        POP       DS            ; restore Seg-reg.
        POP       BP            ; rest. old frame
        RETF      0A            ;  POP 10 Bytes

CRC     ENDP
;
        END
```

Listing 6.8: CRC.ASM

Die Einbindung des OBJ-Moduls in Turbo Pascal 4.0-7.0 ist dann recht einfach. Wichtig ist, daß die Prozedur als FAR aufgerufen wird und die Übergabeparameter korrekt definiert wurden. Das nachfolgendes Listing zeigt beispielhaft wie dies realisiert werden kann.

```
{***********************************************
```

```
  File       : DEMO.PAS
  Vers.      : 1.1
  Autor      : G. Born
  Files      : ---
  Progr. Spr.: Turbo Pascal 4.0 (und höher)
  Betr. Sys. : DOS ab 2.1
  Funktion: Das Programm dient zur Demonstration
  des Aufrufes von Assemblerprogrammen aus Turbo
  Pascal. Es wird das Programm CRC.OBJ einge-
  bunden. Die zu übertragenden Zeichen stehen als
  Bytes im Feld buff[].
***********************************************}

TYPE Buffer = Array [1 .. 255] OF Byte;
VAR crc_res : Word;                { CRC Register }
    buff : Buffer;                 { Datenpuffer  }

{*************       Hilfsroutinen **************}

{*
    Hier wird die OBJ-Datei eingebunden
    und die Prozedur definiert
*}
{$L CRC.OBJ}                      { OBJ. File    }
{$F+}                             { FAR Modell ! }
procedure CRC (var crc_res : word; var buff : Buffer; len :
integer);
external;
{$F-}

procedure Write_hex (value, len : integer);
 {
   Ausgabe eines Wertes als Zahl auf dem Bildschirm.
   Durch Len wird festgelegt, ob ein Byte (Len = 1)
   oder Wort (Len = 2) ausgegeben werden soll.
  }
 const Hexzif : array [0..15] of char = '0123456789ABCDEF';
       Byte_len = 1;
       Word_len = 2;

 TYPE zahl = 1..2;
 VAR temp : integer;
    carry : zahl;
     i    : zahl;
 begin
  if len = Word_len then
   begin
    temp := swap (value) and $0FF;      { high byte holen     }
    write (Hexzif[temp div 16]:1,Hexzif[temp mod 16]:1);
   end;
  temp := value and $0FF;               { low byte holen      }
  write (Hexzif[temp div 16]:1,Hexzif[temp mod 16]:1);
end;  { Write_hex }

{**** Hauptprogramm ****}

begin

 crc_res := 0;                          { clear CRC - Register }
```

```
 buff[1] := $55;                          { Testcode setzen      }
 buff[2] := $88;
 buff[3] := $CC;

 writeln ('CRC - Demo (c) Born G.');
 writeln;
 writeln ('CRC-Berechnung per Polynomdivision');
 writeln;

 CRC (crc_res, buff, 3);                  { Aufruf CRC Routine 1 }
 write ('Die CRC - Summe ist : ');
 write_hex (crc_res,2);                   { Hexzahl ausgeben     }
 writeln;
end.                                      { Ende                 }
```

Listing 6.9: DEMO.PAS

Anschließend läßt sich das Programm als EXE-File mit der Eingabe:

DEMO

aufrufen.

Um das Programm CRC.OBJ in QuickBasic 4.x einzubinden, geht man analog vor. Das nachfolgende Listing in QuickBasic demonstriert die Einbindung des OBJ-Moduls.

```
'!*******************************************************
'! File       : DEMO.BAS
'! Vers.      : 1.1
'! Autor      : G. Born
'! Files      : ---
'! Progr. Spr.: Quick Basic 4.0 / 4.5
'! Betr. Sys. : DOS 2.1 - 4.01
'! Funktion: Das Programm dient zur Demonstration der Ein-
'!           bindung von Assemblermodulen in QuickBasic.
'!           Es wird das Modul CRC.OBJ benutzt, Die Parameter
'!           stehen als Bytes im Feld buff%(), oder im String
'!           buff$. Aus diesen Zeichen wird dann die CRC16-
'!           Summe mittels der Procedur CRC (File CRC.OBJ)
'!           berechnet. Compiler und Linker sind mit folgenden
'!           Parametern aufzurufen:
'!
'!           BC DEMO.BAS
'!           LINK DEMO.OBJ,CRCA.OBJ
'!
'!*******************************************************

DIM buff%(255)                  '! Integer Puffer

crcres% = 0                     '! clear CRC-Register
'!
'! setze Zeichen in Puffer, beachte aber, daß es kein Byte
'! Datum gibt, d.h. zwei Bytes sind in einer Integer Variablen
'! zu speichern !!!
```

```
'!
buff%(0) = &H8855                '! setze Zeichen in
buff%(1) = &HCC                  '! INTEGER Puffer
'!
'! Setze Zeichen alternativ in den Stingpuffer
'!
buff$ = CHR$(&H55)+CHR$(&H88)+CHR$(&HCC) '! Testcode setzen

PRINT "CRC-Demo Programm in Basic (c) Born G."
PRINT
PRINT "CRC-Berechnung per Polynomdivision"
PRINT
'!
'! Berechne CRC aus Integer Puffer
'!
CALL CRC (SEG crcres%, SEG buff%(0), BYVAL 3) '! Aufruf CRC
Routine 1

PRINT "Die CRC - Summe ist : ";HEX$(crcres%) '! Hexzahl ausgeben

'!
'! Berechne CRC aus String Puffer
'!
crcres% = 0                      '! clear CRC-Register
FOR i% = 1 TO LEN(buff$)         '! separiere Zeichen
 tmp% = ASC(MID$(buff$,i%,1))    '! in Integer wandeln
 CALL CRC (SEG crcres%, SEG tmp%, BYVAL 1) '! CRC Routine 1
NEXT i%

PRINT
 '! Hexzahl ausgeben
PRINT "Die CRC - Summe ist : ";HEX$(crcres%)
END
'!**** Ende *****
```

Listing 6.10: DEMO.BAS

Die Einbindung in andere Programmiersprachen kann analog erfolgen. Gegebenenfalls sind die Compilerhandbücher als Referenz zu benutzen.

Damit möchte ich die Ausführungen zu TASM beenden. Sicherlich konnten nicht alle Aspekte des Produkte vorgestellt werden. Hier ist die entsprechende Originaldokumentation des Herstellers zu konsultieren. Für einen ersten Einstieg sollte der vorliegende Text aber genügen.