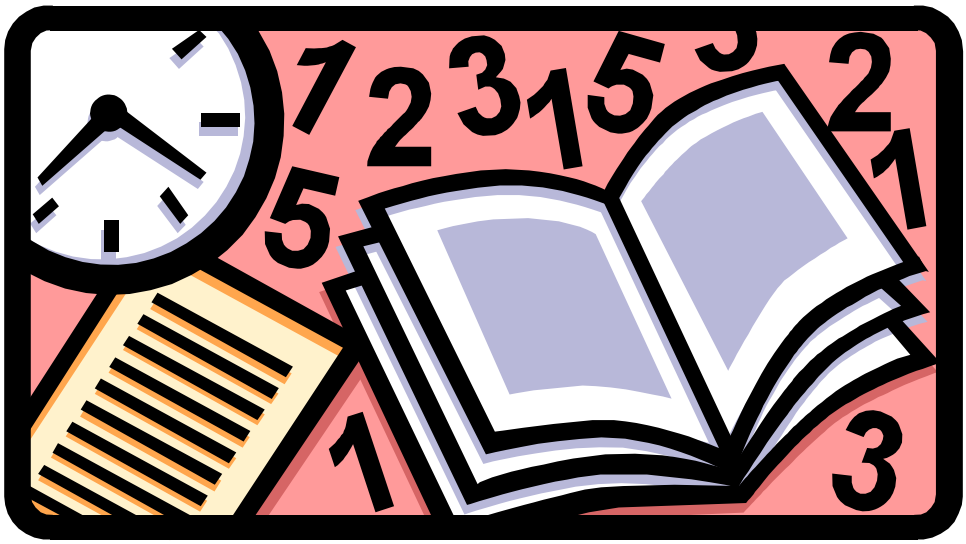
Assembler - Einführung 2. Teil

Assembler – Tools und Umgebung



1. Programmaufbau

Wie im ersten Teil schon erwähnt wird ein Assemblerprogramm in verschiedene Segmente aufgeteilt. Ein Assemblerprogramm enthält in der Regel mindestens drei Segmente :

- STACK
- Datensegment
- Codesegment

Wir haben nun bei der direkten Segmentierung die Möglichkeit diese Segmente persönlich anzulegen und nötigenfalls die Segment- Startadressen in die entsprechenden Zeiger- und Indexregister (DS, CS) zu schreiben.

Als Alternative steht bei neueren Assemblern noch die indirekte Segmentierung über Speichermodelle zur Verfügung. Dies ist interessant, wenn die 64- KB- Barriere der Segmente überwunden werden muss. Die einzelnen Speichermodelle können hierzu ein Programm teilweise in beliebig viele Segmente aufteilen. Es stehen insgesamt 6 Speichermodelle zur Verfügung.

1. Direkte Segmentierung

Ein Segment wird im Quellcode in folgender Form dargestellt:

```
Name segment        [Ausrichtung] [Komb] [Klasse]  ; Beginn des Segmentes.
                                                   ; Diese
                                                   ; Zeilen
                                                   ; sind der Inhalt des Segmentes.
[Name] ends                                        ; Ende des Segmentes.
```

Bei den rot geschriebenen Wörtern handelt es sich um Assemblerbefehle. Diese müssen in genau dieser Form eingesetzt werden. Bei den grün geschriebenen Wörtern handelt es sich um einen notwendigen Zusatz, der das Segment von anderen Segmenten im Programm unterscheidbar macht. Diese sind aber wahlfrei. Man kann sich etwas „schönes“ ausdenken ☺. Es ist immer empfehlenswert Kommentare im Programmcode zu platzieren. Vor diesen Kommentaren muss jedoch ein „;“ stehen.

2. Das erste Programm

Nun währen wir in der Lage ein erstes Programm zu schreiben.
Es soll – das ist so Tradition – "Hallo Welt ! " ausgeben.

Zu diesem Zweck öffnen wir unter Windows ein DOS- Fenster. Hierzu klickt man „START“ → „PROGRAMME“ → „MS- DOS- EINGABEAUFFORDERUNG“.

Es erscheint ein DOS Fenster mit dem Inhalt:
C:\Windows> _

wir geben ein : CD.. und drücken die RETURN – Taste. Es erscheint folgende Ausgabe:
C:\> _

wir geben ein : MD ASM und drücken die RETURN – Taste. Es erscheint folgende Ausgabe:
C:\> _

Jetzt haben wir ein Verzeichnis mit dem Namen ASM angelegt. In dieses Verzeichnis wollen wir nun wechseln. Dazu geben wir ein: CD ASM und drücken die RETURN – Taste.
Es erscheint folgende Ausgabe:
C:\asm> _

Jetzt geben wir ein : EDIT HALLO.ASM und drücken die RETURN – Taste.

Es öffnet sich als blaues Fenster der DOS- Editor. Hier schreiben wir nun unseren Programmcode in die Datei „HALLO.ASM“. Die Dateiendung „ASM“ ist wichtig !

Andere Schreibprogramme sind problematisch, da sie in der Regel nicht nur den Text an sich in der Datei abspeichern, sondern noch zusätzlich Steuerungszeichen für die Darstellungsweise des Textes.

Diese Steuerungszeichen drehen dann beim Assemblieren des Textes den Assembler durch den Wind. Die so verursachten Fehlermeldungen sind geeignet einen an den Rand des Nervenzusammenbruchs zu bringen, da man im Quellcode einfach keinen Fehler finden kann.

Wir geben ein:

```
; ********************************************
;*Ausgabe: „Hallo Welt!“ auf Bildschirm*
;********************************************
;

DATEN SEGMENT

        MELDUNG  DB  "Hallo Welt !“, "$“

DATEN ENDS


STAPEL SEGMENT BYTE STACK

                DW 128 DUP (?)

STAPEL ENDS


CODE SEGMENT

ASSUME CS:CODE,DS:DATEN,ES:NOTHING,SS:STAPEL


START:  MOV AX, DATEN
        MOV DS, AX
        MOV DX, OFFSET MELDUNG


        MOV AH, 9H
        INT 21H

        MOV AH, 4CH
        INT 21H

CODE ENDS

      END START
```

Nun muss unser Quellcode noch gespeichert werden. Hierzu fahren wir mit der Maus im DOS Fenster auf "DATEI" → "SPEICHERN" → "BEENDEN".

Wir sehen wieder auf dem Bildschirm im DOS Fenster den Eingabe Prompt:
C:\asm> _

Jetzt müssen zwei Programme TASM.EXE (der Assembler) UND TLINK.EXE (der Linker) in das Verzeichnis ASM kopiert werden. Das sind die Programmiertools von BORLAND. Von Microsoft gibt es diese Programme auch. Dort haben sie die Bezeichnungen MASM.EXE und LINK.EXE. Bei der Erstellung von anspruchsvolleren Programmen sollte man immer darauf achten, dass man die aktuellste Version dieser Programme hat. Auch bei der Syntax der jeweiligen Assembler gibt es kleine Unterschiede zwischen BORLAND und Microsoft. Die hier vorgestellten Programme wurden mit TASM und TLINK erstellt.

Jetzt geben wir ein : TASM HALLO.ASM und drücken die RETURN – Taste.

Wenn Du keine Tippfehler in Deinem Quellcode hast und auch keine ";" vergessen hast, müsste nun folgende Meldung erscheinen:

TURBO ASSEMBLER VERSION .....

ASSEMBLER FILE : HALLO.ASM

ERROR MESSAGES : none
WARNING MESSAGES : none
…..
C:\asm> _

Jetzt geben wir ein : TLINK HALLO.OBJ und drücken die RETURN – Taste.

folgende Meldung erscheinen:
TUROBO LINK VERSION.....
C:\asm> _

Jetzt geben wir ein: HALLO.EXE und drücken die RETURN – Taste.

folgende Meldung erscheinen:
HALLO WELT !
C:\asm> _

Nun haben wir endlich das erste Programm zum laufen gebracht ! Mit der Hochsprache C++ hätte man einfach COUT << "Hallo Welt !" eingegeben.

Nun wollen wir uns einmal anschauen wie unser Programm vom Assembler übersetzt wird. Dazu geben wir nun ein: TASM / ZI HALLO,, und drücken die RETURN – Taste.
Es wurde dadurch eine Datei HALLO.LST erstellt. Diese Datei wird nun im DOS Editor durch die Eingabe von : EDIT HALLO.LST und drücken der RETURN – Taste geöffnet.

Es erscheint folgende Ausgabe:

```
  1                                     ;***********************************************
  2                                     ;*Ausgabe : "Hallo Welt !" auf Bildschirm*
  3                                     ;***********************************************
  4
  5
  6   0000                              DATEN SEGMENT
  7
  8
  9   0000  48 61 6C 6C 6F 20 57 +                 MELDUNG DB "Hallo Welt !","$"
 10         65 6C 74 20 21 24
 11
 12   000D                              DATEN ENDS
 13
 14
 15   0000                              STAPEL SEGMENT BYTE STACK
 16
 17   0000  80*(????)                                   DW 128 DUP (?)
 18
 19   0100                              STAPEL ENDS
 20
 21
 22
 23   0000                              CODE SEGMENT
 24
 25                                     ASSUME CS:CODE,DS:DATEN,ES:NOTHING,SS:STAP
L
 26
 27   0000  B8  0000s                   START:  MOV AX, DATEN
 28   0003  8E  D8                             MOV DS, AX
 29   0005  BA  0000r                          MOV DX, OFFSET MELDUNG
 30
 31   0008  B4  09                             MOV AH, 9H
 32   000A  CD 21                              INT 21H
 33
 34   000C  B4  4C                             MOV AH, 4CH
 35   000E  CD 21                              INT 21H
 36
 37   0010                              CODE ENDS
 38
 39                                          END START

Symbol Name                            Typ         Value

??DATE                                 Text        "25/03/00"
??FILENAME                             Text        "HALLO  "
??TIME                                 Text        "21:31:19"
??VERSION                              Number      0101
@CPU                                   Text        0101H
@CURSEG                                Text        CODE
@FILENAME                              Text        HALLO
@WORDSIZE                              Text        2
MELDUNG                                Byte        DATEN: 0000
START                                  Near        CODE: 0000

Groups  & Segments                     Bit     Size    Align   Combine     Class

CODE                                   16      0010    Para    none
DATEN                                  16      000D    Para    none
STAPEL                                 16      0100    Byte    Stack
```

Im oberen Ausdruck können wir schön erkennen, wie sich der Assembler die Anordnung unseres Programms im Arbeitsspeicher vorgestellt hat.

In der ersten Spalte werden Zeilennummern angegeben. Von diesen macht der Assembler Gebrauch, wenn er uns Fehlermeldungen um die Ohren haut. Hinter einer Fehlermeldung befindet sich immer die Zeilennummer in der sich der Fehler befindet. Diese Zeilennummern sind nicht Bestandteil unseres Programms im Arbeitsspeicher.

Zeilen 1,2,3 sind Kommentarzeilen. Der Text hinter dem Semikolon befindet sich nur zur Information eines nachfolgenden Programmierers im Quellcode. Auch ein nachfolgender Programmierer soll wissen worum es in diesem Programm geht. Dieser Text ist nicht Bestandteil unseres Programms im Arbeitsspeicher.

Zeilen 4,5 sind Leerzeilen.

In Zeile 6 teilen wir dem Assembler mit, dass wir nun ein Segment (Bereich im Arbeitsspeicher) beginnen wollen. Der Assembler antwortet uns mit dem Hinweis, dass er ab dem nächsten verfügbaren Paragraphenbyte dieses Segment mit Namen DATEN beginnt und die erste Speicherstelle in dem Segment DATEN mit der Offset- Adresse 0000 adressiert.

In Zeile 9 teilen wir dem Assembler mit, dass eine Zeichenkette Hallo Welt ! im Segment DATEN des Arbeitsspeichers abgelegt werden soll. Der Rechner kann aber keine Buchstaben sondern nur Zahlen im Arbeitsspeicher ablegen. Also wandelt er gemäß dem ASCII Code die Buchstabe in Zahlen um. Man kann zum Beispiel gut erkennen, dass die hexadezimale Zahl 20 für ein Leerzeichen steht. Da 13 Zeichen inklusive Leer- und $-zeichen abgelegt werden sollen und jedes Zeichen 1 Byte (1 Speicherplatz) belegt, befinden wir uns am Ende der Zeichenkette an Speicherplatz 000D (= 0013 dezimal). Außerdem teilen wir mit MELDUNG dem Assembler mit, dass wir der Speicherstelle 0000 den Namen MELDUNG zuordnen, und dass wir an anderer Stelle des Programms durch Verwendung des Namens MELDUNG auf diese Speicherstelle zugreifen wollen. Mit DB (DEFINE Byte) geben wir zu verstehen, dass Speicherplätze nur byteweise belegt werden sollen

In Zeile 12 Teilen wir dem Assembler mit dem ENDS Befehl mit, dass hier dieses Segment endet. Der Assembler bestätigt dies mit dem Hinweis, dass für ihn damit an der Offset- Adresse 000D dieses Segment beendet ist.

Für Zeile 15 gilt entsprechend das Gleiche wie für Zeile 6. Der Zusatz BYTE bedeutet, dass das Segment an der nächsten Adresse beginnen soll. Der Zusatz STACK zeigt an, dass wir nur mit SS:SP auf diesen Bereich zugreifen wollen. Ehrlich gesagt brauchen wir den STACK in diesem Programm gar nicht. Aber 1. kann man daran erkennen wie man einen STACK anlegt und 2. ist es sehr einfach das Reservieren von Speicherplatz zu demonstrieren.

In Zeile 17 hätten wir nun der Startadresse wieder einen Namen, etwa MAKE_PROG_LOOK_BIG, zuordnen können. Da wir aber eh nicht auf diesen Speicherplatz zugreifen wollen, wäre dies überflüssig. Der Ausdruck DW steht für DEFINE Word. Mit dieser Direktive geben wir an, dass wir Speicherplatz immer nur als Vielfaches von 2 Byte (=1 Word) belegen wollen. Die Zahl 128 bewirkt, dass 128 solcher Einheiten (eine Einheit besteht aus je 2 Byte) im Arbeitsspeicher reserviert werden. Insgesamt werden also 256 Byte im Arbeitsspeicher reserviert. Damit wird klar warum das Segment STAPEL an Adresse 0100 (hexadezimal) in Zeile 19 endet. Die hexadezimale Zahl 0100 entspricht der Dezimalzahl 256. Das "?" mit der DUP ( ) Direktive bewirkt, dass die einzelnen Speicherplätze mit keinem Standartwert vorbelegt werden. Ihr Inhalt bleibt so, wie wir ihn vorgefunden haben. Die hexadezimale Zahl 80 entspricht der Dezimalzahl 128.

In Zeile 23 beginnt das Segment CODE.

In Zeile 25 müssen wir nun dem Prozessor mitteilen, welche Segmentadressen nun den einzelnen Allzweckregistern zugeordnet werden. Im CS- Register landet die Segmentadresse vom Codesegment, weil dort die Befehle abgelegt sind usw.

In Zeile 27 kopieren (bewegen) wir die Segmentadresse des Datensegmentes in das CPU Register AX.

In Zeile 27 kopieren (bewegen) wir die Segmentadresse des Datensegmentes nun von AX in das Datensegmentregister DS.
Eine Vereinfachung zu MOV DS, DATEN ist nicht möglich, da der MOV Befehl dies nicht zulässt.

In Zeile 29 bewirkt der Ausdruck OFFSET MELDUNG, dass die Speicherstelle mit dem Namen Meldung in das Register DX kopiert wird. Zusätzlich kommt zum Ausdruck, dass diese Speicherstelle nur den Beginn eines Speicherbereiches darstellt. Damit kann der Prozessor nun unsere Zeichenkette HALLO WELT ! finden. Im DX Register steht die Adresse an der, der String beginnt. Da es aber viele Speicherstellen mit der Adresse 0000 im Programm gibt, teilen wir ihm zusätzlich im DS Register mit in welchem Segment der Prozessor die Adresse suchen soll.

Die Zeilen 27 und 28 sind immer notwendig, wenn wir auf Adressen im Datensegment zugreifen wollen!!!

Warum verschieben wir die Startadresse unseres Strings gerade in das DX Register ?
Die Antwort auf diese Frage steht in Zeile 32.

Dort wird mit dem Befehl INT unser Programm unterbrochen um ein Unterprogramm des Betriebssystems mit der "Nummer" 21 zu starten. Dieses Unterprogramm ist Bestandteil des Betriebssystems und wurde mit ihm geliefert.

Dieses Unterprogramm schaut immer zuerst in das Register AH und sieht nun dort die hexadezimale Zahl 9H (Das H steht für hexadezimal). Das Unterprogramm weiß dadurch, dass es eine Zeichenkette auf dem Bildschirm ausgeben soll. Das Unterprogramm verlangt, dass die Adresse der ersten Speicherstelle dieser Zeichenkette im Register DX steht. Das Unterprogramm nimmt allerdings automatisch an, dass diese Speicherstelle sich in dem Segment befindet, dessen Adresse in DS angegeben ist. Jetzt gibt das Unterprogramm nacheinander den Inhalt der Speicherstellen auf dem Bildschirm aus. Wenn das Unterprogramm jedoch auf das $-zeichen trifft, signalisiert dies dem Unterprogramm, dass die Zeichenkette zu Ende ist und es seine Arbeit beenden kann. Nun gibt das Unterprogramm die Kontrolle an unser Programm zurück.

In Zeile 35 rufen wir schon wieder dieses Unterprogramm auf. Mit der Zahl 4CH im Register AH teilen wir diesmal dem Unterprogramm mit, dass unser Programm zu Ende ist. Das Unterprogramm gibt jetzt die Kontrolle nicht mehr an unser Programm sondern ans Betriebssystem zurück. Wir sehen den DOS- Prompt auf dem Bildschirm.

In Zeile 39 Befindet sich der Ausdruck END START. Das Label hinter END signalisiert dem Assembler an welcher Stelle der erste Befehl in unserem Programm zu finden ist (bei uns in Zeile 27). Jetzt weiß der Prozessor welche Startadresse er in sein Register IP laden muss (nämlich die Offset- Adresse von START= 0000).

In Zeile 27 steht in der 3. Spalte 0000s. Das "s" bedeutet, dass dies eine Segmentadresse sein soll. In Zeile 29 steht in der 3. Spalte 0000r. Das "r" steht hier für Relativadresse (Offset- Adresse). Wie kommt der Assembler auf die Idee dem Datensegment die Adresse 0000 zugeben ? Was ist, wenn zu Beginn des Arbeitsspeicher (am ersten Paragraphenbyte) schon ein anderes Programm seine Daten geparkt hat ? Die Antwort: Der Assembler hat gar keinen Einfluss darauf, wo unser Programm im Arbeitsspeicher abgespeichert wird. Die "Parkplatzeinweisung" macht der Lader des Betriebssystems. Er kennt die Adresse des letzten noch freien Speicherplatzes. Sollte zum Beispiel die letzte freie Speicherstelle die Adresse 0400H haben, so addiert der Lader einfach zu jeder Segmentadresse in unsrem Programm 0400H dazu und legt das Programm an dieser Stelle im Arbeitsspeicher ab.

Übrigens: Dies ist auch der Grund warum man Adressen nicht addieren darf !

3. DEBUG

Eines der wichtigsten Hackertools ist mit Sicherheit das Programm DEBUG. Es ermöglicht einen Blick hinter die Kulissen. Außerdem ist es das einzige Programm mit dem man an jedem Rechner mit MS-DOS und in allen Lebenslagen ein Programm schreiben kann
(Vorausgesetzt man versteht etwas von Assembler) . Dies ist möglich, weil DEBUG mit MS- DOS geliefert wird. Die Programme haben jedoch die Einschränkung, dass sie nur aus einem Segment bestehen können. Da wir aber mit DEBUG an fremden Rechnern keine Programme schreiben wollen, für die Mann- Jahre notwendig sind, ist dies meistens auch kein Problem.

3.1 DEBUG als Testhilfe

Ja, dann nehmen wir mal unser Programm Hallo.exe auseinander.

Dazu öffnen wir wieder ein DOS- Fenster und wechseln in das Verzeichnis ASM.
Wir sehen:
C:\asm> _

Wir geben ein:
DEBUG HALLO.EXE und drücken die RETURN – Taste.

Wir sehen:
```
C:\asm>debug hallo.exe
-_
```

Wir geben ein: t und drücken die RETURN – Taste.

Wir sehen:
```
C:\asm>debug hallo.exe
-t
AX=1BA5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1B95  ES=1B95  SS=1BA5  CS=1BB6  IP=0003   NV UP EI PL NZ NA PO NC
1BB6:0003 8ED8          MOV     DS,AX
-
```
DEBUG zeigt uns in den ersten beiden Zeilen die Inhalte der Register unserer CPU nach Ausführung der Zeile 27 in obigem Ausdruck. In der dritten Zeile sehen wir die nächste zur Ausführung vorgesehene Programmzeile. Außerdem sehen wir nun in welchen Bereich des Arbeitsspeichers der Lader das Programm geladen hat. Die Adressen der jeweiligen Segmente sind hier blau unterlegt.

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=1BA5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=0005   NV UP EI PL NZ NA PO NC
1BB6:0005 BA0000        MOV     DX,0000
-
```

Es ist klar, dass bei jedem Rechner andere Segmentadressen als in diesem Beispiel auftreten werden. Der Arbeitsspeicher ist je nach Rechner unterschiedlich groß und es werden vorher unterschiedliche Programme in den Speicher geladen.

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=1BA5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=0008   NV UP EI PL NZ NA PO NC
1BB6:0008 B409          MOV     AH,09
-
```

DEBUG arbeitet im Einzelschritt Modus unser Programm ab. Man sieht deutlich, dass im Register IP schon immer die Adresse des nächsten Befehls steht.

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=09A5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=000A   NV UP EI PL NZ NA PO NC
1BB6:000A CD21          INT     21
-
```

Wie man sieht steht nun tatsächlich im hohen Teil von AX eine 09, die wir ein Rechenschritt zuvor dorthin "gemoved" haben.
Wir sind nun wieder an die Stelle gekommen, an der das Unterprogramm "Nr.21H"des Betriebssystems aufgerufen wird. Schauen wir uns doch mal an wohin die Reise im Arbeitsspeicher geht.

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=09A5  BX=0000  CX=0120  DX=0000  SP=0107  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1194  IP=0445   NV UP DI PL NZ NA PO NC
1194:0445 EAA004B407    JMP     07B4:04A0
-
```

Hä? Was ist das ? Wir sind an der Speicherstelle 1194:0445 gelandet ! Dies ist aber noch nicht unser Unterprogramm. Wir müssen einen Sprung (JMP) zur Adresse 07B4:04A0 ausführen, um zum Unterprogramm zu gelangen. Wir sind in einer Tabelle gelandet. Jede Zahl bei einem INT Befehl führt zu einer Speicherstelle an der lediglich die Adresse des eigentlichen Unterprogramms steht.

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=09A5  BX=0000  CX=0120  DX=0000  SP=0107  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=07B4  IP=04A0   NV UP DI PL NZ NA PO NC
07B4:04A0 80FC72        CMP     AH,72
-
```

Hier sehen wir die erste Zeile des Unterprogramms. Nach unserem INT Befehl haben wir unser Programm verlassen und haben sozusagen das Hoheitsgebiet von Microsoft betreten. Wir könnten uns auf diese Weise den Quellcode des ganzen Unterprogramms zugänglich machen und die geistigen Ergüsse der Microsoft Programmierer begutachten. In der Fachsprache nennt man das zugänglich machen des Quellcodes eines Programms Disassemblierung. Das Unterprogramm ist lang und langweilig. Bevor wir unseren Rechner abschießen

geben wir diesmal ein: q und drücken die RETURN – Taste.

Dadurch wurde DEBUG verlassen und wir sehen den DOS Prompt. Wir fangen wieder von vorne (3.1 DEBUG) an bis wir wieder an die Stelle kommen :

```
AX=09A5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=000A   NV UP EI PL NZ NA PO NC
1BB6:000A CD21          INT     21
-
```

Wir geben diesmal an dieser Stelle ein p ein und drücken die RETURN – Taste.

Nun wird das gesamte Unterprogramm am Stück durchgeführt. Daher sehen wir nun:

```
Hallo Welt !
AX=0924  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=000C   NV UP EI PL NZ NA PO NC
1BB6:000C B44C          MOV     AH,4C
-
```

und landen wieder in unserem Programm.

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=4C24  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=000E   NV UP EI PL NZ NA PO NC
1BB6:000E CD21          INT     21
-
```

Wir geben ein: p und drücken die RETURN – Taste.
Wir sehen:

```
Programm wurde normal beendet
-
```

Wir geben ein: q und drücken die RETURN – Taste.
Wir sehen:

```
C:\asm>_
```

3.2 Editieren des Arbeitsspeichers mit DEBUG

Jetzt haben wir bis auf unseren String HALLO WELT ! alles von unserem Programm im Arbeitsspeicher gesehen. Und den String werden wir uns nun auch noch anschauen.

Zu diesem Zweck müssen wir erst einmal herausfinden wo sich der String im Arbeitsspeicher befindet.

1. **Schritt:**

Wir öffnen wieder ein DOS- Fenster und wechseln in das Verzeichnis ASM.
Wir sehen:
C:\asm> _

Wir geben ein:
DEBUG HALLO.EXE und drücken die RETURN – Taste.

Wir sehen:

```
C:\asm>debug hallo.exe
-_
```

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
C:\asm>debug hallo.exe
-t

AX=1BA5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1B95  ES=1B95  SS=1BA5  CS=1BB6  IP=0003   NV UP EI PL NZ NA PO NC
1BB6:0003 8ED8          MOV     DS,AX
-
```

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=1BA5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=0005   NV UP EI PL NZ NA PO NC
1BB6:0005 BA0000        MOV     DX,0000
-
```

Wir geben ein: t und drücken die RETURN – Taste.
Wir sehen:

```
AX=1BA5  BX=0000  CX=0120  DX=0000  SP=010D  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA5  CS=1BB6  IP=0008   NV UP EI PL NZ NA PO NC
1BB6:0008 B409          MOV     AH,09
-
```

Nun haben wir den String HALLO WELT ! in den Arbeitspeicher geladen. Ohne diesen Schritt wäre es Zufall, was wir an den entsprechenden Speicherstellen antreffen.

2. **Schritt:**

Der String HALLO WELT ! steht bei mir an der Adresse 1BA5: 0000 Bei Dir ergibt sich hier mit Sicherheit für den vorderen Teil eine andere Adresse. Der String HALLO WELT ! benötigt mit Leerzeichen und $-zeichen 13 Speicherstellen. Das Ende des String befindet sich dann an der Adresse1BA5 : 000D.
.
Wir geben ein: E 1BA5:0000 und drücken die RETURN – Taste.
Wir sehen:

```
-E 1BA5:0000
1BA5:0000  48._
```

Mit der hexadezimalen Zahl 48 sehen wir hier das "H" von HALLO. Wir haben nun 3 Möglichkeiten:

1) Durch drücken der Leerzeichen- Taste ohne Veränderung des Wertes zum nächsten Zeichen vorzurücken.
2) Einen neuen Wert einzugeben und dann mit Drücken der Leerzeichen- Taste zum nächsten Wert vorzurücken.
3) Durch drücken der RETURN Taste den Editiermodus zu verlassen ohne etwas zu verändern.

Ich habe mich gerade dafür entschieden aus unserm Programm die englischsprachige Version HELLO WORLD! zumachen.

Da das H noch stimmt drücken wir nun die Leerzeichentaste.
Wir sehen:

```
-E 1BA5:0000
1BA5:0000  48.    61._
```

Das "e" ist im ASCII Code ist einfach 4 Buchstabe weiter. Das bedeutet wir ersetzten die Zahl 61 durch die Zahl 65.

Wir geben ein: 65 und drücken die LEERZEICHEN – Taste.
Wir sehen:

```
-E 1BA5:0000
1BA5:0000  48.    61.65   6C._
```

Wir drücken so oft die LEERZEICHEN- Taste bis wir an diesen Punkt kommen:

```
-E 1BA5:0000
1BA5:0000  48.    61.65  6C.    6C.    6F.    20.    57.    65._
```

Hier steht nun das "e" von Welt. Wir ersetzten es durch ein "o" → Zahl: 6F

wir tippen ein: 6F und drücken die LEERZEICHEN – Taste.
Wir sehen:

```
1BA5:0000  48.    61.65  6C.    6C.    6F.    20.    57.    65.6F
1BA5:0008  6C._
```

Hier steht nun das "l" von Welt. Wir ersetzten es durch ein "r" → Zahl: 72

wir tippen ein: 72 und drücken die LEERZEICHEN – Taste.
Wir sehen:

```
1BA5:0000  48.    61.65  6C.    6C.    6F.    20.    57.    65.6F
1BA5:0008  6C.72  74._
```

Hier steht nun das "t" von Welt. Wir ersetzten es durch ein "l" → Zahl: 6C

wir tippen ein: 6C und drücken die LEERZEICHEN – Taste.
Wir sehen:

```
1BA5:0000  48.    61.65  6C.    6C.    6F.    20.    57.    65.6F
1BA5:0008  6C.72  74.6C  20._
```

Hier steht nun das " " hinter dem Wort Welt. Wir ersetzten es durch ein "d" → Zahl: 64

wir tippen ein: 64 und drücken die LEERZEICHEN – Taste.
Wir sehen:

```
1BA5:0000  48.    61.65  6C.    6C.    6F.    20.    57.    65.6F
1BA5:0008  6C.72  74.6C  20.64  21._
```

Wir sehen nun den ASCII Code des Ausrufezeichens. Der bleibt unverändert.

wir drücken die LEERZEICHEN – Taste.
Wir sehen:

```
1BA5:0000  48.    61.65  6C.    6C.    6F.    20.    57.    65.6F
1BA5:0008  6C.72  74.6C  20.64  21.    24._
```

Wir sehen nun den ASCII Code des $-zeichens. Der bleibt unverändert.

wir drücken die RETURN – Taste.
Wir sehen:
-
Wir geben ein: g und drücken die RETURN – Taste.
Wir sehen:

```
Hello World!
Programm wurde normal beendet
```

-
Wir geben ein: q und drücken die RETURN – Taste.
Wir sehen:

```
C:\asm>_
```

Wenn wir nun hallo eingeben und RETURN drücken kommt wieder die Meldung Hallo WELT !.
Dies liegt daran, dass das Programm mit den alten Werten von der Festplatte neu geladen wird.

3.3 Erstellen von ausführbaren Programmen mit DEBUG

Wir schreiben nun direkt ein Programm in den Arbeitsspeicher und lassen es anschließend mit DEBUG assemblieren und ausführen. Diese ausführbaren Dateien können (wegen DEBUG) nur aus einen Segment bestehen. Dies bedeutet, dass wir die Daten und den Programmcode in das **gleiche** Segment schreiben müssen. Daher müssen wir dafür Sorge tragen, dass der Prozessor nicht auf unsren String "Hallo Welt !" trifft und denkt das seien Befehle die er jetzt ausführen muss.
DEBUG setzt nämlich alle Segmentregister auf den gleichen Wert. Dieses Problem lässt sich dadurch entschärfen, indem man die Daten im richtigen Moment einfach überspringt.

Dazu öffnen wir wieder ein DOS- Fenster und wechseln in das Verzeichnis ASM.
Wir sehen:
C:\asm> _

Wir geben ein: DEBUG und drücken die RETURN – Taste.

Danach wird ein a eingegeben und die RETURN – Taste gedrückt.

Wir sehen (Die Segmentadresse ist wahrscheinlich eine andere):

1B72:0100 _

Da wir als erstes den String "Hallo Welt !" abspeichern wollen müssen wir zuerst einen Sprung eingeben:

Jetzt brauchen wir ein wenig Kopfrechnen. Der Sprungbefehl mit Adresse benötigt 2Byte im Speicher. Überspringen wollen wir den String "Hallo Welt !","$". Dafür brauchen wir noch einmal 13 Byte. Der Befehl MOV AH, 09H könnte erst im 14 Byte abgespeichert werden. Da müssen wir mit dem JMP Befehl hinspringen.
Das bedeutet: Speicherplatz 0100 für JMP

```
                0101 für  Sprungadresse
                0102      H
                    .
                    .
                010E      $
                010F      MOV AH, 09H
                    .
                    .
```

Wir springen also zu 010F

Wir geben ein: JMP 010F und drücken die RETURN – Taste.
Wir sehen:

```
-a
1B72:0100 JMP 010F
1B72:0102_
```

Wir geben ein: DB "Hallo Welt !","$" und drücken die RETURN – Taste.
Wir sehen:

```
-a
1B72:0100 JMP 010F
1B72:0102 DB "Hallo Welt !","$"
1B72:010F_
```

Tatsächlich landen wir am richtigen Platz!

Wir geben ein: MOV AH, 9 und drücken die RETURN – Taste.
Wir sehen:

```
-a
1B72:0100 JMP 010F
1B72:0102 DB "Hallo Welt !","$"
1B72:010F MOV AH,9
1B72:0111_
```

Das Unterprogramm möchte bekanntlich im DX Register die Offset- Adresse des Strings haben.
Der String beginnt in Speicherstelle 0102. Diese Speicherstelle laden wir nun nach DX.

Wir geben ein: MOV DX, 102 und drücken die RETURN – Taste.
Wir sehen:

```
C:\asm>debug
-a
1B72:0100 JMP 010F
1B72:0102 DB "Hallo Welt !","$"
1B72:010F MOV AH,9
1B72:0111 MOV DX, 102
1B72:0114_
```

Wir geben nun den restlichen Text ein, bis wir diese Stelle erreichen:

```
-a
1B72:0100 JMP 010F
1B72:0102 DB "Hallo Welt !","$"
1B72:010F MOV AH,9
1B72:0111 MOV DX, 102
1B72:0114 INT 21
1B72:0116 MOV AH, 4C
1B72:0118 INT 21
1B72:011A_
```

Nun wird nichts mehr eingegeben sondern einfach nur die RETURN – Taste gedrückt.
Wir sehen:

```
-a
1B72:0100 JMP 010F
1B72:0102 DB "Hallo Welt !","$"
1B72:010F MOV AH,9
1B72:0111 MOV DX, 102
1B72:0114 INT 21
1B72:0116 MOV AH, 4C
1B72:0118 INT 21
1B72:011A
-_
```

Wir geben ein: n WELT.COM und drücken die RETURN – Taste.
Wir geben ein: RCX und drücken die RETURN – Taste.
Wir sehen:

```
-n WELT.COM
-RCX
CX 0000
:_
```

DEBUG möchte jetzt, dass wir die Anzahl der Bytes eingeben sie unser Programm belegt.
Bei uns sind dies 25 Bytes.

Wir geben ein: 25 und drücken die RETURN – Taste.

Wir geben ein: w und drücken die RETURN – Taste.

Wir sehen:
CX 0000
:25
-w
00025 Bytes werden geschrieben
-_

Wir geben ein: g und drücken die RETURN – Taste.
Wir sehen:

Hallo Welt !
C:\asm>_

Wir haben auf diese Weise eine COM Datei geschrieben. Der Unterschied zur EXE Datei besteht darin, dass die COM Datei nur 64 KB groß sein kann und aus einem Segment besteht.

3.4 Disassemblierung mit DEBUG

Mit dem Parameter u kann man sich das Programm Hallo.exe im Arbeitsspeicher anzeigen lassen. DEBUG erkennt aber nicht wo unser Programm zu Ende ist.

```
Microsoft(R) Windows xx
  (C)Copyright Microsoft Corp 19xx-19xx.

C:\WINDOWS>cd..

C:\>cd asm

C:\asm>debug hallo.exe
-u
1BB6:0000 B8A51B      MOV     AX,1BA5
1BB6:0003 8ED8        MOV     DS,AX
1BB6:0005 BA0000      MOV     DX,0000
1BB6:0008 B409        MOV     AH,09
1BB6:000A CD21        INT     21
1BB6:000C B44C        MOV     AH,4C
1BB6:000E CD21        INT     21
1BB6:0010 3C2A        CMP     AL,2A
1BB6:0012 7503        JNZ     0017
1BB6:0014 83CA02      OR      DX,+02
1BB6:0017 3C3F        CMP     AL,3F
1BB6:0019 7503        JNZ     001E
1BB6:001B 83CA04      OR      DX,+04
1BB6:001E 0AC0        OR      AL,AL
-
```

Es gibt natürlich noch andere Programme zum Disassemblieren (CODEVIEW) die wesentlich komfortabler sind als DEBUG. Diese Einführung erhebt keinen Anspruch auf vollständige Darstellung aller Möglichkeiten und Sachverhalte. Sie soll lediglich einen Eindruck vom grundsätzlichen Vorgehen vermitteln.