



Neuigkeiten:

- Neu: Aktuelle (26.12.07) Homepage **jetzt** als PDF-Datei! Auch für Firefoxnutzer die Probleme haben die kleine Code-Schrift zu lesen ==> [PDF Hier \(242 KB\)](#)
- Neu: Buch nun in der 2. Auflage erhältlich! Siehe weiter unten...
- Neu: Neues Homepage-Design. Dank an Camal Cakar, der u.a. das Banner gestaltet hat!
- Neu: Mirror unter <http://www.kaskade.gmxhome.de/asm/> Schneller und ohne Werbebanner auf der eigentlichen Seite!
- Neu: Zwei neue Programme (vom 07.01.05): 32Bit-Zahlenausgabe (mit Vorzeichen) + Win32-Programm "DSOLink" [hier!](#).

Infos wie man an einen kostenlosen Microsoft-Assembler rankommt: [Hier!](#)

Ganz neu: Das Buch Assembler Grundlagen der Programmierung. Mehr als 100 Seiten sind dazugekommen! [Hier mehr Informationen dazu!](#) Erhältlich ist es überall im Buchhandel, sowie bei Amazon.de, Buecher.de usw. 600 Seiten, mit Programmierung unter DOS (von mir: [Marcus Roming](#)) und Windows (von Joachim Rohde). Infos zur ersten Auflage (Tipps und Errata) [gibt es hier!](#)
Erschienen im MITP Verlag für 29.95 Eur. ISBN:3-8266-1469-0

Einführung:

Vorweggenommen: Assembler ist entgegen seinem Ruf nicht superkompliziert. Man kann damit sehr systemnah programmieren und ist nicht eingeschränkt. Der vom Assembler erzeugte Code ist im Vergleich zu dem von Hochsprachen a) schneller und b)

deutlich kleiner. Ich werde versuchen den Text möglichst übersichtlich und gegliedert zu halten. Beginnen werde ich mit einer kurzen Einführung zur Funktionsweise des 8086 Prozessors. Keine Angst: der 8086 Prozessor ist zwar veraltet aber alle neuen Prozessortypen bauen darauf auf, d.h. Code der für einen 8086'er geschrieben wurde, läuft selbstverständlich auch auf einem 286'er, 386'er, 486'er, Pentium und was da so noch kommt und kommen mag. Der Unterschied zwischen einem 8086'er und einem Pentium besteht im wesentlichen darin, dass der Pentium Prozessor schneller ist und über mehr Befehle verfügt. Nach der Einführung zur Funktionsweise des 8086 kommt eine kurze Zusammenfassung des Binär bzw. Hexadezimalsystemes. Danach das erste kleine Programm, die wichtigsten Befehle und so weiter.

Bevor das Ganze also losgeht werde ich erstmal sagen, was man alles an Programmen und Infos benötigt um loslegen zu können.

Das wäre im einzelnen:

1. Natürlich ein Assembler. Der Assembler ist ein Programm, welches ihre symbolischen Befehle in eine Zahl bzw. Zahlenfolge d.h. in Maschinenbefehle übersetzt. Der Beste dürfte der Turbo Assembler von Borland sein (TASM) es gibt ihn in der Version 4.0 für ca. 50 DM auf einer CD vom Franzis Verlag ISBN: 3-7723-9442-b. Auf der CD sind noch viele Infos und viel Sourcecode... feine Sache. Auch der Microsoft Assembler (MASM) ist nicht schlecht, die Bedienung fast gleich. Gibt es [hier!](#)
2. Für spätere, eigene Projekte: Ralf Brown's legendäre Interruptliste. Bekommt man zusammen mit einer Portliste von Wim Osterhold, einer Liste zur DOS Speicherbelegung von Robin Walker und einer Interruptzusammenfassung von Bent Lynggaard: Zwei ZIP's (techn. Gründe)
[Teil I \(441 KByte\)](#) & [Teil II \(417 Kbyte\)](#) Für eine aktuellere Version wende man sich an die im jew. File angegebenen Adressen bzw. an die in der Datei Rbrown.txt unter "Availability:" genannten Adressen.
3. Die DOS Technical Reference von Dave Williams ist auch nicht schlecht und beinhaltet zum Teil Info's die die anderen Texte nicht haben. Auch sind ziemlich alle Funktionen von Int 21h (dem wichtigsten) beschrieben. [Download](#) (156 KB)
--> Für den Anfang besser !
4. Ein einfacher wissenschaftlicher Taschenrechner, zur Umrechnung von [Hex]adezimal oder [Bin]är nach Dezimal usw. Hat wahrscheinlich jeder, wenn nicht gibt es gute um 30 DM oder halt im Kopf umrechnen.
5. Last but not least: Meine HTML Files zum [Download](#) (228 KB Zip) Enthält alles ausser der DOS-Technical-Ref., Hed und Ralph Browns Int. Liste. Aktualisiert: 20.03.04 !

Die Homepage als [PDF-Datei!](#) Vom 22.03.05. 240 KB (Dank an HTMLDOC!)

Achtung: Die Listen von 2. und 3. sind für später gedacht, wenn Sie eigene Programme schreiben wollen. Für den Anfang bekommen Sie natürlich alle Info's die Sie brauchen von mir !

Weiter geht es nun mit einer Einführung in den [8086 Prozessor](#).

Sie können sich auch das [Inhaltsverzeichnis](#) ansehen !

*



Der 8086 Prozessor ist wie bereits erwähnt der Vorgänger heutiger Prozessoren wie der 80486'er (kurz 486'er) Prozessor sowie des Pentium bzw. Pentium II Prozessors. Der 8086'er war der erste Prozessor, der 1 MB Speicher adressieren konnte, was damals noch recht viel war. Prinzipiell gibt es nur wenig Unterschiede zwischen einem 8086 und z.B. einem 80486'er. Der 486'er hat eine höhere Taktfrequenz (z.B. 66 MHz statt 8 MHz) und benötigt pro Befehl weniger Takte. Auch verfügt er über grössere Register (32 statt 16 Bit), einen breiteren Daten und Adress-Bus und einen integrierten Coprozessor (zumindest der 486 DX)... Hilfe ! Was zur Hölle ist ein Register oder was ist die Taktfrequenz und das andere Zeug werden Sie jetzt vielleicht sagen. Nun genau das möchte ich hier kurz erläutern. Zuerst werde ich erklären was ein Bit und was ein Byte ist. Der Computer kann eigentlich nur zwischen zwei Zuständen unterscheiden, nämlich 1 und 0 d.h. soviel wie Strom oder nicht Strom. Ein Bit stellt also die kleinste Speichereinheit dar. Aufgrund der Prozessorarchitektur können jedoch Daten nicht in beliebiger Grösse verarbeitet werden. Da die internen Verbindungswege (Bus) aus mindestens 8 Leitungen bestehen, haben die Entwickler von Intel beschlossen, 8 Bits zu einer Einheit zusammenzufassen: Das Byte. Ein Byte ist also gleich 8 Bits.

Da ein Bit entweder 0 oder 1 speichern kann, also 2 Zustände annehmen kann, kann ein Byte 2^8 also 256 Zustände annehmen oder Zahlen von 0 bis 255 entsprechen. Das Bit das am weitesten links steht, ist Bit Nummer 7. Es ist das sogen. höherwertigste Bit. Das Bit rechts aussen ist Bit Nummer 0, ist das niederwertigste Bit. Es ergibt sich also folgende Abfolge:

höherwertigstes Bit ---> 7 6 5 4 3 2 1 0 <--- niederwertigstes Bit.

Zwei Byte ergeben ein Wort (engl. Word). Dies teilt sich auf in zwei Byte - Teile:

Word: [(15 14 13 12 11 10 9 8) (7 6 5 4 3 2 1 0)]
höherwertiges Byte niederwertiges Byte

Analog dazu gibt es noch das Doppelwort (Doubleword=4Bytes), das Quadwort (Quadword=8Bytes) und noch ein paar grössere, die aber nicht so wichtig sind.

Um grössere Speichersummen anzugeben verwendet man noch Kilobyte, Megabyte und Gigabyte: 1024 Byte sind ein Kilobyte (KB), 1024 Kilobyte sind ein Megabyte (MB) und 1024 Megabyte sind ein Gigabyte.

Und nun zu Thema Taktfrequenz. Die Taktfrequenz wird meist in Megahertz (MHz) angegeben. Ein Hertz ist eine Schwingung pro Sekunde, ein Megahertz entspricht folglich einer Million Schwingungen pro Sekunde. Je höher die Taktfrequenz desto mehr Arbeitseinheiten in der Sekunde kann der Prozessor verarbeiten und desto schneller ist der Informationsfluss und somit der Computer. Jeder Maschinenbefehl braucht mindestens ein Takt meist jedoch mehr. Je moderner der Prozessor desto weniger Takte werden pro Befehl benötigt. So benötigt z.B. der Befehl NOP, das ist der Befehl, der im Prinzip nichts tut (No Operation), auf einem 8086'er noch 3 Takte, auf einem 486'er benötigt er 1 Takt. Moderne Prozessoren (Pentium aufwärts) können auch zwei Befehle gleichzeitig ausführen, jedoch sind nur bestimmte Kombinationen zugelassen.

Zum Schluss kommt das wichtigste: Die Register. Register sind Speicher, welche sich im Prozessor befinden, und daher extrem schnell sind. Der Nachteil ist, dass sie klein sein müssen, um in den Prozessor zu passen. Bis zum 286'er gibt es 16 Bit Register danach 32 Bit Register, welche immer ein E vor dem Namen haben (z.B. EAX 32 Bit, AX 16 Bit). Für den Anfang werde ich jedoch nur mit 16 Bit Registern arbeiten, der Einfachheit halber. Man kann die Register in vier Kategorien unterteilen: Allzweckregister, Segmentregister, Zeiger und Indexregister, Flagregister.

1. Allzweckregister:

Die Allzweckregister haben eine Besonderheit: Sie lassen sich in zwei Teile unterteilen, einen High- und einen Low-Teil von jeweils 8 Bit größe. Im Falle des schon genannten AX Registers nennt man die beiden Teile AH und AL. Zu den Allzweckregistern zählen weiterhin BX, CX und DX. Die Register lassen sich als ganzes oder getrennt in High- und Low-Teil bearbeiten. Erklärung der einzelnen Allzweckregister:

a) Das AX Register:

Der Akkumulator (AX Register) wird bei arithmetischen Operationen eingesetzt. Es muss auch bei der Multiplikation und Division verwendet werden.

b) Das BX Register:

Das Base (BX) Register wird bei Speicherzugriffen als Zeiger verwandt.

c) Das CX Register:

Das Count (CX) Register wird zum Beispiel in Schleifen eingesetzt. Beispiel: Eine Schleife mit dem Befehl LOOP wird so lange wiederholt, bis CX = 0 ist.

d) Das DX Register:

Das Data (DX) Register dient zum Beispiel zur Aufnahme von Daten bei 16 Bit Multiplikationen und Divisionen.

2. Segmentregister:

Bei den Segmentregistern handelt es sich um die Register CS (Codesegmentregister), DS (Datensegmentregister), SS (Stacksegmentregister) und ES (Extrasegmentregister). Diese Register werden dazu benötigt, die 1 MB Speicher im Real Mode anzusprechen. Mit den CPU internen 16 Bit Registern kann man jedoch nur 65536 Byte ansprechen, um ein MB anzusprechen wäre ein 20 Bit Register notwendig, das man damals aber aus technischen Gründen nicht bauen konnte. Daher muss man einen Trick verwenden: Man unterteilt das eine MB in Segmente von minimal 16 Byte. Mit einem 16 Bit Register kann man Adressen bis 65536 verwalten, da die Segmente mindestens 16 Byte groß sind ergibt sich $65536 \times 16 = 1 \text{ Megabyte}$! Die Segmentadresse gibt also ein Speicherabschnitt (Segment) innerhalb des 1 MB Adressraumes an. Innerhalb dieser Segmente bewegt man sich mit Hilfe der Offsetadresse. Die Offsetadresse kann wiederum nur maximal 65536 groß sein d.h. die maximale Segmentgröße ist 65536 Byte (64 KB). So lässt sich jede Adresse im physikalischen Speicher durch Angabe der Segment- und der Offsetadresse eindeutig beschreiben. Der Prozessor kann aus diesem Adresspaar (man nennt das Segment-Offset Adresspaar logische Adresse) die 20 Bit physikalische Adresse (also die tatsächlichen Speicherstelle) berechnen.

Dies geschieht durch Multiplikation der Segmentadresse mit 10h (das h steht für Hexadezimal, mehr dazu später 10h=16) und nachfolgender Addition der Offsetadresse.

Beispiel:

Gegeben sei die physikalische Adresse F2003h. Diese Adresse lässt sich logisch z.B. so darstellen: F200h:0003h die Schreibweise ist also: Segmentadresse:Offsetadresse. Rechnet man die logische Adresse nach obiger Regel um, so erhält man F2003h.

Umgekehrt ist dies allerdings nicht eindeutig möglich. Man kann allerdings zur Umrechnung einer physikalischen Adresse in eine logische folgenden Trick anwenden: Man fasst die ersten 4 Ziffern zur Segmentadresse zusammen und bildet aus der letzten Ziffer durch voranstellen dreier Nullen die Offsetadresse.

3. Zeiger und Index Register:

Hier gibt es die Register SI, DI, BP, SP, und IP. Sie lassen sich nur als 16 Bit Register ansprechen.

a) Der Source Index (SI)

Wenn man z.B. einen String (Zeichenkette) verschieben möchte, so muss dieser Register als Zeiger auf den Anfang des Strings dienen (in einem vorgegebenen Segment).

b) Der Destination Index (DI)

Im obigen Fall würde dieses Register das Ziel des zu kopierenden Strings beinhalten.

c) Der Basepointer (BP)

Er wird zum Beispiel verwendet um auf das Stacksegment zuzugreifen.

d) Der Stackpointer (SP)

Der Stackpointer zeigt stets auf die Aktuelle Position im Stacksegment. Der Stack (Stapel) bzw. das Stacksegment wird zur Zwischenspeicherung von Werten benutzt. Zum Teil verwenden bestimmte Prozessorbefehle den Stack (z.B. CALL), aber auch der Benutzer kann den Stack mittels der Befehle push und pop verwenden. PUSH schiebt einen Wert auf den Stack POP holt ihn zurück. Die Daten werden auf dem Stack nach dem Last in – First out (LIFO) Prinzip gespeichert.

Den Befehl, den Sie als letzten gesichert (gepusht) haben, bekommen sie beim poppen als erster wieder heraus. Beispiel: Ich will die Register ax und bx auf dem Stack speichern, und danach beide Register, die vielleicht inzwischen verändert wurden, wieder korrekt herstellen. Dazu wird folgender Code verwendet:

push ax ;speichern von ax

push bx ;speichern von bx

. ;Die Punkte stehen für irgendwelche Befehle

. ;Mit einem ; werden Anmerkungen gekennzeichnet !

pop bx ;letztgespeicherter Wert in bx
pop ax ;erstgespeicherter Wert in ax

Später werde ich noch einmal auf den Stack eingehen !

e) Der Instruction Pointer (IP)

Der Instruction Pointer zeigt immer die momentane Position im Codesegment. Man kann ihn nicht direkt verändern, die Verwaltung übernimmt der Prozessor.

4. Das Flagregister

Das Flagregister ist 16 Bit breit. Es ist als ein Statusregister zu verstehen, und in einzelne Flags unterteilt. Ein Flag kann also entweder gesetzt sein (also = 1) oder nicht (also = 0). Es gibt folgende Flags:

Das Carry-Flag CF (es wird bei einem Überlauf gesetzt oder bei Fehlern), das Auxiliary Flag, das Overflow-Flag, das Sign Flag (es wird für Vorzeichenzahlen genutzt), das Parity Flag, das Direction Flag, das Interrupt Flag, das Trap Flag und das Zero Flag ZF (Das Zero Flag zeigt nach einer Operation an, ob das Ergebnis null ist oder nicht).

Manche dieser Flags sind relativ unwichtig, auf andere werde ich später eingehen.

Zusatzbemerkung: Oben kann man vielleicht den Eindruck haben, dass die Register nur für bestimmte Zwecke gut sind. Das ist nicht so. Hier herrscht frohe Anarchie. So kann man zum Beispiel weil einem danach ist und weil es irgendwie passt den aktuellen Treibstoffverbrauch eines virtuellen Autos in BP (also dem Basepointer) speichern. Das geht gut. Die Allzweckregister sind sowieso für alles gut und die anderen auch, bis auf folgende Ausnahmen: SP sollte nicht direkt verändert werden, IP geht gar nicht direkt, das Flagregister auch nicht.

Die Segmentregister sollten besser Segmentadressen enthalten und sonst nichts.

Ja natürlich ich weiss LOOP nimmt seinen Counter nun mal nur in CX usw. aber das kommt noch und sollte niemanden davon abhalten den Treibstoffverbrauch in BP zu speichern (zumindest im allerschlimmsten Notfall)

So das wäre fürs Erste genug. Als nächstes werde ich kurz auf [Hexadezimale und Binäre Zahlen](#) eingehen.

[Zurück](#) [Weiter](#) [Inhalt](#)



Mit diesem Thema werde ich wie mit allem Mathematischen verfahren: Kurz und schnell. Es gibt schliesslich Taschenrechner. Dennoch sind Grundkenntnisse hier sehr wichtig.

- Das Hexadezimalsystem.

Das hexadezimale Zahlensystem arbeitet mit der Basis 16. Es werden die Ziffern 0–9 verwendet, die restlichen Werte (10 bis 15) werden mit den Buchstaben A,B,C,D,E und F besetzt:

Dezimal : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26...

Hexadezimal: 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C...

Eine hexadezimale Zahl hat meist ein h oder H am Ende. Der Assembler benötigt bei Hex Zahlen welche mit einem Buchstaben beginnen eine Führungsnull, also statt FFh 0FFh ! Die ist nur für den Assembler wichtig, um nichts falsch zu interpretieren.

2. Das Dualsystem.

Das Dualsystem arbeitet mit der Basis 2. Binäre Zahlen werden mit einem b oder B am Ende gekennzeichnet also zum Beispiel 10010001b.

Die Umrechnung von Dual-Zahlen zu Dezimalen-Zahlen geht so:

Gegeben sei die Zahl 10101b Umrechnung:

Ziffer Dual: 1 0 1 0 1
Basis : 2⁴ 2³ 2² 2¹ 2⁰ (^ = hoch)

$$\text{Ziffer dezimal} = 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 1 = 16 + 0 + 4 + 0 + 1 = 21$$

Einfach oder ?

Nun noch die logischen Operatoren:

Logische Operatoren verknüpfen zwei Werte immer Bitweise.

1. Die AND–Verknüpfung:

Bit1	Bit2	AND
0	0	0

1	0	0
0	1	0
1	1	1

--> Nur wenn beide Bits gesetzt sind ergibt die Verknüpfung den Wert 1

2. Die OR-Verknüpfung:

<u>Bit1</u>	<u>Bit2</u>	<u>OR</u>
0	0	0
1	0	1
0	1	1
1	1	1

--> Es kommt 1 heraus, wenn eines oder beide Bits den Wert 1 haben.

3. Die XOR-Verknüpfung (Exklusiv – Oder):

<u>Bit1</u>	<u>Bit2</u>	<u>XOR</u>
0	0	0
1	0	1
0	1	1
1	1	0

--> Nur wenn beide Bits unterschiedliche Werte haben kommt 1 heraus.

NOT invertiert einfach alle Bits.

Nun kommt endlich das erste [Programm](#) !

[Zurück](#) [Weiter](#) [Inhalt](#)



Nun werde ich zeigen wie man ein einfaches Assemblerprogramm schreibt. Es geht mir vor allem darum, zu zeigen wie ein Assemblerprogramm aufgebaut ist und wie man es assembliert und linkt. Das erste Programm wird eine EXE Datei sein, die lediglich den Text "Hallo Welt !" auf dem Bildschirm anzeigt. Das Programm "Hallo Welt" ist als erstes Programm eigentlich Standard, und auch ich werde mich daran halten.

Um das Programm zu erstellen benötigen Sie lediglich einen ASCII Editor, am besten das DOS-Programm Edit. Die Datei sollte die Erweiterung asm haben. Erstellen Sie also z.B. eine neue Datei mit dem Namen first.asm indem sie im DOS folgendes eingeben: edit first.asm und dann Return oder Enter drücken.

In die Datei selbst schreiben Sie dann Folgendes mit Ausnahme des Textes hinter dem ;, denn der sagt dem Assembler, dass ein Kommentar folgt, und den können Sie weglassen:

```
.MODEL SMALL      ;Das ist das Speichermodell, das verwendet wird
.STACK 100h       ;Die grössse des Stack's festlegen
.DATA             ;Beginn des Datensegments

Meldung           DB  "Hallo Welt!$" ;Die Meldung selbst

.CODE
mov  ax,@data     ;Adresse des Datensegments (@data) nach ax
mov  ds,ax        ;Die Adresse des Datensegments in das Segmentregister

mov  dx,OFFSET Meldung ;Die Offsetadresse von Meldung nach dx
mov  ah,09h       ;Den Wert neun nach ah schieben
int  21h          ;Interrupt 21h Funktion 09h aufrufen
                    ;= Write String. Geschrieben wird bis zum $
mov  ah,4Ch       ;DOS Funktion Programm beenden
int  21h          ;Programm beenden

END              ;Anweisung an den Assembler.
```

Das Programm selbst ist momentan noch Nebensache. Es geht hier vor allem darum, aus diesem Code eine ausführbare EXE-Datei (von engl. execute = ausführen) zu erstellen. Dazu muss die asm Datei erst einmal assembliert werden. Wir erinnern uns: Der Assembler macht aus den symbolischen Anweisungen die Sie gerade eingegeben haben Maschinenbefehle, die der Computer versteht.

Geben Sie also am DOS-Prompt ein: TASM first und drücken Sie Return.
Die Endung .asm wird automatisch angehängt.

Nun Erzeugt der Assembler eine Objektdatei (first.obj).
Auf dem Bildschirm sollten Sie folgendes sehen:

```
Turbo Assembler  Version 4.0  Copyright (c) 1988, 1993 Borland International
```

```
Assembling file:   first.ASM
Error messages:    None
```



```
Warning messages:  None
Passes:           1
Remaining memory:  455k
```

Wenn nicht, überprüfen Sie, ob sie keine Fehler gemacht haben, tut sich gar nichts, überprüfen Sie ob der Pfad ihres Assemblers bzw. Linkers in der PATH Variable der Datei Autoexec.bat steht.

Geben Sie nun am DOS-Prompt ein: TLINK first und drücken Sie Return.
Die Endung .obj wird automatisch anhängt.

Nun Erzeugt der Linker eine EXE-Datei (first.exe aus der Datei first.obj).
Auf dem Bildschirm sollten Sie folgendes sehen:

```
Turbo Link  Version 6.00 Copyright (c) 1992, 1993 Borland International
```

Starten Sie nun First.exe. Sie sollten etwa folgendes sehen:

```
C:\>first
Hallo Welt!
C:\>
```

Sollte dies nicht der Fall sein, dann überprüfen Sie nochmal die asm Datei, und assemblieren und linken diese dann nochmal. Wenn sie den Microsoft Assembler verwenden lauten die Anweisungen MASM first bzw. LINK first ! Infos zu neueren MASM-Versionen [Hier!!](#) Best. Linker können folgenden Fehler erzeugen: fatal: no entry point. Dann muß man den Entrypoint explizit angeben (bei MASM immer!), der Entrypoint ist der Punkt im CODE wo das Programm bei der Ausführung dann startet. Hier die [modifizierte ASM-Datei](#). Sie müssen dann auch alle anderen EXE-Projekte dementsprechend abändern!

Die Mapdatei (first.map) die vom Linker noch erzeugt wird, enthält Informationen über die Segmente des Programmes, schauen Sie einfach mal rein (edit first.map).

Nun werden Sie vielleicht fragen, wieso so umständlich, könnte das nicht ein einziges Programm erledigen ? Nun der Linker ist eigentlich dazu da, mehrere OBJ Dateien zusammen zu linken (engl. link=verbinden). Das ist zum Beispiel dann wichtig, wenn an einem Projekt mehrere Menschen arbeiten. In unserem Fall ist es nur eine OBJ Datei.

Nun werde ich die Befehle von oben anhand des Programmes genauer [erklären](#).

[Zurück Weiter Inhalt](#)



Ich habe bei dem Programm die vereinfachten Segmentanweisungen benutzt, welche nur von neueren Assemblern unterstützt werden. Ich weiss nicht wie verbreitet ältere Assembler sind, möglicherweise werden diese Anweisungen von ihrem Assembler nicht unterstützt. Ich werde daher die normalen Segmentanweisungen im Anhang kurz erklären. Programmiert habe ich im MASM Modus, da ich vom Ideal Modus nicht viel halte. Der MASM Modus hat auch den Vorteil, dass er sowohl vom MASM als auch von TASM verstanden wird.

Das Assemblerprogramm beginnt mit **.MODEL SMALL**. Dieser Befehl richtet sich an den Assembler selbst, und wird nicht direkt übersetzt. Der Befehl legt fest, dass Daten und Code jeweils ein eigenes Segment erhalten, dass nicht grösser als 64 KB sein darf. **.MODEL TINY** würde bedeuten, dass Daten und Code in ein einziges Segment passen müssen, das ebenfalls nicht grösser als 64 KB sein darf. Dies würde dann, zusammen mit ein paar anderen Modifikationen eine COM Datei (engl. compile = zusammenstellen) erzeugen.

Die nächste Anweisung ist **.STACK 100h**. Dies legt die grösse des Stack auf 256 Byte fest, wenn man versucht mehr darauf zu speichern gibt es einen Stack overflow oder Stapelüberlauf. Versucht man etwas vom Stack zu holen, wenn dieser leer ist gibt es einen Stack underflow. Für unser Programm ist ein Stack mit 256 Byte mehr als gross genug. Wird kein Wert angegeben werden 1024 Byte reserviert.

.DATA definiert das Datensegment. Hier kommen Daten und Variablen, Puffer usw. rein. Die Daten werden durch ein Label gekennzeichnet, welches vom Assembler in eine Adresse umgesetzt wird. In unserem Fall heisst das Label "Meldung". Der Namen kann beliebig gewählt werden, jedoch werden nur die ersten 31 Zeichen berücksichtigt. Folgende Zeichen sind in Labels erlaubt: 1. A–Z 2. a–z 3. 0–9 (nicht am Anfang des Namens) 4. _ und @ und ? und \$ (dürfen nicht allein stehen). Zwischen Gross und Kleinbuchstaben wird nicht unterschieden.

DB steht für define Byte. Der Assembler zählt automatisch die Zeichen in den doppelten Hochkommata (es gehen auch einfache) und reserviert den Speicherplatz. Wenn sie doppelte Hochkommata benutzen, dürfen sie diese nicht in der Zeichenkette benutzen und andersherum. D.h. so geht es: DB "Tom's cat" und so: DB 'Er sagte: "Hallo"'.

.CODE bezeichnet den Anfang des Codesegments. D.h. nun folgen die Befehle.

Der Befehl **mov** kopiert den Inhalt des Quelloperanden in den Zieloperanden. So schiebt z.B. **mov ax,1500** den Wert 1500 nach AX. Also ist danach AX = 1500. **@data** enthält die Adresse des Datensegments (der Assembler ermittelt diese Zahl). **mov ax,@data** schiebt also die Adresse des Datensegments nach ax. **mov ds,ax** schiebt die den Wert von AX nach DS. Das Datensegmentregister enthält also nun die Adresse des Datensegments.

mov ds,@data sieht zwar kürzer aus, geht aber nicht, da man Segmentregister nicht direkt mit Konstanten laden kann. Man muss also den Umweg über ein Universalregister in Kauf nehmen. **mov dx,OFFSET Meldung** schiebt die Offsetadresse von Meldung nach dx. **OFFSET Meldung** wird vom Assembler in einen Wert umgesetzt, **OFFSET** selbst ist kein Maschinenbefehl.

mov ah,09h schiebt den Wert 9 nach ah. AX wird sozusagen zur Hälfte überschrieben. **int 21h** verzweigt in ein Unterprogramm des Betriebssystems. Das Programm wird unterbrochen (Interrupt!) und DOS führt nun seine Funktion 09h aus: Write String, schreibe Zeichenkette. Die DOS-Funktion muss in ah stehen (in unserem Fall ist es Funktion 9). Ausserdem benötigt die DOS-Funktion noch

Angaben über den Ort der Zeichenkette. Dieser wird durch die Segmentadresse in DS und die Offsetadresse in DX definiert. Das Ende der Zeichenkette wird dem Betriebssystem durch ein \$ mitgeteilt, welches selbst nicht angezeigt wird.

Das Unterprogramm welches durch Int 21h ausgelöst wurde kehrt am Ende wieder in das Programm zur nächsten Anweisung zurück. Alle nötigen Informationen hat der Prozessor im Stack gespeichert. **mov ah,4Ch** schiebt 4Ch nach AH. Die Kennung der Interruptfunktion befindet sich also nun in ah. Int 21h löst ein weiteres Unterprogramm aus, welches das Programm beendet. **END** zeigt dem Assembler das Ende an, hier wird nicht mehr weiterübersetzt.

Nun geht es [weiter](#) mit genaueren Informationen zum Aufbau eines Assemblerprogramms.

[Zurück Weiter Inhalt](#)



Definition: Assembleranweisungen.

Eine Anweisungszeile weist den Assembler an, etwas zu tun, was mit der Erzeugung von Befehlen nichts zu tun hat. Die Anweisungen sind z.B. zur Deklaration von Daten und Datenstrukturen da, oder beeinflussen die Arbeitsweise des Assemblers.

1. Format einer Assembler-Zeile:

[Label] [Befehl/Anweisung] [Operanden] [;Kommentar]

Die eckigen Klammern sollen zeigen, dass die aufgeführten Komponenten optional sind, man kann sie also auch weglassen. Der Abstand zwischen den einzelnen Komponenten kann beliebig gross sein (>0), er sollte jedoch aufgrund der besseren Übersichtlichkeit nicht zu gross gewählt werden. Auch sollten die Spalten einheitlich gestaltet werden.

2. Das Label:

Das Label kennzeichnet Variablen, Werte und Sprungziele. Beim assemblieren werden die Labels in Adressen umgesetzt.

Beispiele:

```
Var1 dw ? ;Anlegen einer Variable, von 2 Byte (dw = define Word) Var1 steht für die
        ;Adresse, an der der Assembler den Platz für die Variable Var1 reserviert.
```

```
Beenden:                ;Diese Label dient als Sprungziel, es benötigt daher am
        mov ah,4Ch        ;Ende ein Doppelpunkt
        int 21h
```

3. Variablen und Konstanten.

Variablen kommen normalerweise in das Datensegment. Nur in COM Dateien, die ich später noch erklären stehen sie im Codesegment. Variablen sind mindestens ein Byte gross bzw. das ganzzahlige Vielfache eines Bytes, also z.B. ein Wort (2 Byte) oder ein Doppelwort.

Durch die Assembleranweisungen DB (define Byte), DW (define Word), DD (define Doubleword), DQ (define Quadword) und DT (define Tenbytes) kann man Speicherplatz im Datensegment reservieren. Dieser kann dann über den Variablennamen angesprochen werden. Beispiel:

```
Im Datensegment:
Variable1 DB 40h ;Variable1 ist mit dem Wert 40h vorbelegt.
```

```
Im Codesegment:
mov ah,Variable1 ;Den Inhalt der Variablen nach ah schieben.
```

```
mov dx,OFFSET Variable1 ;Die Offsetadresse der Variablen nach dx schieben.
```

a) Uninitialisierte Variablen.

Oft bekommt eine Variable erst im Verlauf des Programmes einen Wert und muss daher nicht mit einem Wert vorbesetzt werden. Um eine solche uninitialisierte Variable zu erstellen, setzt man anstelle des Wertes ein Fragezeichen :

Variable2 DB ? ;reserviert ein uninitialisiertes byte.

b) Zeichenketten oder Strings.

Zeichenketten beinhalten zum Beispiel den Text, der auf den Bildschirm ausgegeben wird, oder in eine Datei geschrieben wird. Da ein Zeichen ein Byte benötigt, verwendet man DB. Der String selbst steht in zwei " oder ´.
Siehe auch vorherige Seite!

Beispiel:

```
Text1 DB "So geht´s "
Text2 DB ´Und "so" auch´
Text3 DB "So "aber" nicht"      ;<-- Fehler !!
Text4 DB ´So geht´s auch nicht´ ;<-- Fehler !!
```

c) Zahlen.

Natürlich können Variablen auch numerische Werte haben.
So erzeugt zu Beispiel die folgende Assembleranweisung

```
Zahlen DB 1,2,3,4,5,6
```

einen 6 Bytes langen Speicherbereich, wobei das erste Byte den Wert 1 hat, das zweite Byte den Wert 2 usw.

Die Assembleranweisung

```
Zahl DW 888
```

reserviert 2 Bytes und speichert darin die Zahl 888.

d) Der DUP-Operator.

Um ein Tabelle mit sich wiederholenden Werten zu erstellen, kann man die Assembleranweisung DUP verwenden. Um z.B. eine Tabelle mit 12 Nullen zu erzeugen, kann man entweder schreiben:

```
Tabelle1 DB 0,0,0,0,0,0,0,0,0,0,0,0
```

Oder aber viel kürzer:

```
Tabelle1 DB 12 DUP (0)
```

Weitere Beispiele:

Tabelle DB 2 DUP (1,2,3)	entspricht:	Tabelle DB 1,2,3,1,2,3
Tabelle DB 9 DUP (?)	entspricht:	Tabelle DB ?,?,?,?,?,?,?,?,?
Tabelle DW 4 DUP (888)	entspricht:	Tabelle DW 888,888,888,888

e) Konstanten.

Konstanten brauchen selbst keinen Speicherplatz, eine Konstante enthält einen Wert, der im Programm nicht verändert werden kann. Die Konstante wird gewöhnlich am Anfang definiert. Die Konstante kann dann im Programm anstatt ihres Wertes verwendet werden.

Eine Konstante wird mit der Assembleranweisung EQU definiert: *Name EQU Ausdruck*.

Beispiel:

```
                .MODEL SMALL
                .STACK 100h

;Definition der Konstanten
```

```

Write      EQU    09h
Beenden    EQU    4Ch

                .DATA
Msg         DB     "Hallo$"

                .CODE
mov  ax,@data
mov  ds,ax
mov  dx,OFFSET Msg
mov  ah,Write      ;Write=09h !
int  21h
mov  ah,Beenden    ;Beenden=4Ch !
int  21h
END

```

Auch ist zum Beispiel folgendes möglich:

```

ProgramSize  EQU ProgramEnd - ProgramStart

```

Wenn ProgramEnd ein Label am Ende des Programms ist und ProgramStart eines am Anfang, den enthält ProgramSize die Gesamtlänge des Programms.

Konstanten machen die Wartung des Programmes wesentlich einfacher, da anstatt vieler Werte nur einer geändert werden muss.

Weiter geht es mit [wichtigen Befehlen](#).

Diese sollten Sie in jedem Fall durchlesen, um einen Überblick zu gewinnen.

[Zurück Weiter Inhalt](#)



==> Die für den Anfang besonders wichtigen Befehle wurden mit einem roten "!" versehen! Die Liste wurde um einige Opcodes ergänzt, welche in der Regel am Ende der Beschreibung in eckigen Klammern stehen. Hat ein Befehl mehr als ein Opcode, dann werden diese mit einem ; getrennt. Alle Opcodes sind Hex-Werte! (Ein Opcode ist quasi der übersetzte Assembler Befehl)

ADD

Addiere beide Operanden und lege das Ergebnis in den Ersten.

Beispiel:

```
mov  ah,7
mov  bh,3
add  bh,ah    ;bh = 10
```

AND

Logische UND-Verknüpfung: Der Zieloperand wird bitweise mit dem Quelloperand verknüpft.

AND Ziel,Quelle. Das Ergebnis wird im Zieloperand abgelegt.

Beispiel:

```
mov  ah,10010100b    ;ah=10010100b
mov  al,10110101b    ;al=10110101b
and  ah,al            ;ah=10010100b
```

!CALL

Aufruf eines Unterprogramms. Der IP wird auf dem Stack gespeichert, dann wird der IP mit der Adresse des CALL-Befehls geladen, RET kehrt vom Unterprogramm zurück.

Beispiel:

```
.
.
call Write            ;Das Unterprogramm Write aufrufen
.
.
Write PROC  Near      ;Deklaration des Unterprogramms
mov  ah,09h           ;Beispielcode
int  21h
ret                   ;Zurück
Write ENP             ;Ende der Prozedur
```

CALL ist das Gleiche wie CALL NEAR.

Mit CALL FAR Kann man Prozeduren aufrufen, die nicht im selben Segment liegen.

Bei CALL FAR muss man RETF verwenden.

[E8;FF;9A]

CLC

Clear Carry Flag d.h. CF = 0. Das Carry Flag wird oft als Schalter verwendet, z.B um Fehler in einem Unterprogramm anzuzeigen. Mit CLC kann das Carry vorher gelöscht werden.

CLD

Clear Direction Flag: Setzt das Direction Flag auf 0. Damit erfolgen alle Stringoperationen (z.B. stosb) von links nach rechts (default!!).

CLI

Lösche das Interruptflag. Die extern maskierbaren Interrupts werden gesperrt.
D.h. es werden z.B Interrupts die vom Keyboard oder von der Maus erfolgen gesperrt.
Dies ist vor allem dann erforderlich, wenn Zeitkritische Operationen ablaufen, die nicht gestört werden sollen.

!CMP

Mit CMP werden zwei Operanden verglichen. Die Operanden müssen gleich gross sein, d.h. entweder Byte/byte oder Word/word. Das Ergebnis des Vergleichs kommt ins Statusregister.
Beispiel:

```
mov ax,0FFFFh
cmp ax,Var1
je Gleich      ;Wenn gleich springe zum Label Gleich
jg Grösser    ;Wenn grösser springe zum Label Grösser
jl Kleiner    ;Wenn kleiner springe zum Label Kleiner
[3A;3B;3C;3D;38;39;80;81;83]
```

CMPS

CMPSB (Compare Strings Bytes) / CMPSW (Compare Strings Words)

Mit dem Befehl REP /REPNE können mehrere Daten verglichen werden.

Der Befehl CMPS vergleicht zwei Strings miteinander, wobei der erste Operand über das Registerpaar DS:SI und der zweite Operand über ES:DI adressiert wird. Nach dem Vergleich werden SI und DI automatisch erhöht oder bei gesetztem Direction Flag erniedrigt. Das Ergebnis des Vergleichs kommt ins Statusregister.

Beispiel:

```
lea di,String1      ;Wenn di auch im Datensegment einfach ES=DS z.B. so: push ds pop es
lea si,String2      ;lea si,String2 entspricht mov si,OFFSET String2
mov cx,12            ;Anzahl der Wiederholungen von REP
rep cmpsb            ;Vergleichen
je Weiter            ;Springen wenn gleich
```

!DEC

Decrementieren um 1. D.h. es wird 1 vom Operanden subtrahiert.
Gegenteil: INC.

Beispiel:

```
mov ah,09h
dec ah      ;ah=08h
```

Oder als LOOP Ersatz (glaub sogar schneller):

```
      mov cx,4
Schleife:
      dec cx
      .
      .
      jnz Schleife
```

Oder um eine Variable zu Dekrementieren:

```
dec Var1
```

DIV

Dividiert AX bzw. das Registerpaar DX:AX durch den Divisor.

Divident	/	Divisor	=	Quotient	Rest
AX (16 Bit)	/	Operand (8 Bit)	=	AL	AH
DX:AX	/	Operand (16 Bit)	=	AX	DX

Beispiel:

```
mov ax,300      ;AX = Divident
mov bh,12       ;bh = Divisor
div bh          ;AX/BH = AL(Quotient) AH(Rest)
```

HLT

Der Befehl HLT (HaLT) hält den Prozessor an bis ein Interrupt erfolgt.

IN

In liest ein Byte oder Word aus einem Port in al oder ax ein.

Beispiel:

```
in  al,60h    ;Byte vom Tastaturpuffer in al lesen
```

INC

Addiert (Incrementiert) den Operanden um 1.

Beispiel:

```
mov  ax,07h
inc  ax      ;ax=08h
inc  Var1    ;Incrementieren einer Variablen im Speicher
```

!INT

Löst einen Interrupt aus. Wenn ein Interrupt über den INT Befehl ausgelöst wird passieren zwei Dinge:

1. Das Flagregister wird auf den Stack gepusht
2. Ein FAR CALL wird ausgelöst zu der Segment:Offset Adresse die sich in der Interruptvektor-Tabelle befindet (dort umgekehrt abgespeichert Offset:Segment). Bei dieser Adresse befindet sich der Code des Interrupts. Die Rückkehr ins Programm erfolgt durch einen IRET Befehl.

[CD; Speziell Int 3: CC]

Diese Tabelle besteht aus 256 Doppelwörtern und reicht von 0000:0000 bis 0000:0400h.

Die Offset-Adresse der Interruptvektoren kann man errechnen indem man die Interruptnummer mit 4 multipliziert. Die Segmentadresse ist immer 0000h. So ist z.B. die Adresse für den Interrupt 21h in der Tabelle an der Stelle $21h * 4 = 84h$ also 0000:0084h.

IRET

Rückkehr aus einer Interruptroutine. CS:IP und das Statusregister (Flags) wird vom Stack genommen.

!JMP

Unbedingter Sprung zur angegebenen Adresse bzw. Label.

JMP FAR kann auch zu Zielen springen, die nicht im eigenen Segment liegen.

Beispiel:

```
        jmp    Weiter
        .
        .
        .
```

Weiter:

[EB;E9;FF;EA]

!Bedingte Sprünge:

JE/JNE/JZ/JNZ/JC/JNC/JG/JL/JLE/JGE sind wichtige bedingte Sprünge.

Sie werden nur ausgeführt, wenn eine bestimmte Bedingung erfüllt ist.

JE: Jump if equal. Springe wenn gleich.

Beispiel:

```
cmp  ax,Var1    ;vergleicht die Variable Var1 und ax indem beide subtrahiert werden.
je   Gleich     ;Sind sie gleich gross, ist das Ergebnis 0 und das Zero Flag wird
                    ;gesetzt. JE prüft dieses Flag und springt wenn es gesetzt also 1 ist.
```

JE : Jump if equal. Springe wenn gleich. [74;0F84]

JNE : Jump if not equal. Springe wenn nicht gleich. [75;0F85]

JZ : Jump if zero. Springe wenn 0. Eigentlich das Gleiche wie JE, es hat sogar den gleichen Opcode, d.h. es wird in die gleiche Zeichenfolge übersetzt. Aber: Übersichtlicher wenn man nicht nur eins verwendet. [74;0F84]

JNZ : Jump if not zero. Springe wenn nicht 0. [75;0F84]

JC : Jump if Carry. Springe wenn Carry Flag gesetzt. [72;0F82]

JNC : Jump if not Carry. Springe wenn Carry Flag nicht gesetzt. [73;0F83]

JG : Jump if greater. Springe wenn grösser. [7D;0F8D]

JNG : Jump if not greater. Springe wenn nicht grösser. [7E;0F8E]

JL : Jump if less. Springe wenn kleiner. [7C;0F8C]

JNL : Jump if not less. Springe wenn nicht kleiner. [7D;0F8D]

JLE : Jump if less-equal. Springe wenn kleiner oder gleich.

JGE : Jump if greater-equal. Springe wenn grösser gleich.

JNLE: Jump if not less-equal. Springe wenn nicht kleiner oder gleich.
JNGE: Jump if not greater-equal. Springe wenn nicht grösser oder gleich.

Bedingte Sprünge haben lediglich eine Reichweite von -128 bzw. +127 Bytes. Es kann daher schnell passieren, dass ein Ziel ausserhalb der Reichweite liegt. In diesem Fall meldet der Assembler einen Fehler der wie folgt behoben werden kann:

Wenn es so nicht geht:

Anfang:

```
.  
.;Über 128 Bytes Programmcode  
.  
cmp ax,00h  
je Anfang ;Sprungziel zu weit entfernt. --> Fehler  
mov bx,40h
```

Dann geht es so:

Anfang:

```
.  
.;Über 128 Bytes Programmcode  
.  
cmp ax,00h  
jne Weiter1 ;Sprungziel zu weit entfernt. --> Fehler  
jmp Anfang  
Weiter1: mov bx,40h
```

!LEA

LEA Ziel,Quelle (Load Effective Address). LEA berechnet den Offset und speichert das Ergebnis im Zieloperand, d.h. im angegebenen Register.

Beispiel:

```
lea dx,String1  
mov dx,OFFSET String ;hat fast den selben Effekt, LEA ist besser.
```

LODS

Lädt über das DS:SI Register ein Byte in das AL Register (LODSB) oder ein Wort in das AX Register (LODSW). Danach wird das SI Register entsprechend (Byte oder Word) um 1 oder 2 erhöht ist das Direction-Flag gesetzt wird es erniedrigt.

!LOOP

Loop springt zur angegebenen Adresse bis das CX Register null ist. Loop subtrahiert vom CX Register 1 und überprüft CX auf null. Ist CX nicht null springt LOOP zur angegebenen Adresse, wenn LOOP null ist wird das Programm mit dem nächsten Befehl fortgeführt.

Beispiel:

```
mov cx,11 ;Die Schleife wird 11 mal durchlaufen  
DoLoop:  
nop ;zu wiederholende Befehle hier: NOP  
  
Loop DoLoop
```

LOOPE-LOOPZ

Springt zur angegebenen Adresse, wenn CX ungleich null ist und das Zero-Flag gesetzt ist.

Beispiel:

```
mov cx,11 ;Die Schleife wird max. 11 mal durchlaufen  
DoLoop:  
nop ;zu wiederholende Befehle hier: NOP  
cmp ax,Var1  
Loope DoLoop ;Springe wenn cx ungleich null und ax und Var1 gleich
```

LOOPNE-LOOPNZ

Springt zur angegebenen Adresse, wenn CX ungleich null ist und das Zero-Flag nicht gesetzt ist.

Beispiel:

```
mov cx,11 ;Die Schleife wird max. 11 mal durchlaufen
```

```

DoLoop:
    nop          ;zu wiederholende Befehle hier: NOP
    cmp  ax,Var1
    Loope DoLoop ;Sprunge wenn cx ungleich null und ax und Var1 ungleich

```

!mov

Der Befehl MOV Ziel,Quelle (engl. move bewegen) überträgt ein Byte oder Word vom Quelloperanden in den Zieloperanden. Der Quelloperand selbst wird nicht verändert. Quell und Zieloperand müssen gleich gross sein, etwas wie z.B. mov al,cx funktioniert nicht. Auch kann man mit MOV nicht die Register CS, IP und das Flagregister nicht verändern. Mögliche Operandenkombinationen:

```

mov  Reg,Reg      Bsp: mov  cx,ax
mov  Reg,Seg. Reg Bsp: mov  ax,ds
mov  Reg,Mem      Bsp: mov  cx,Variable
mov  Reg,Wert     Bsp: mov  bx,40h
mov  Seg. Reg,Reg Bsp: mov  es,cx
mov  Seg. Reg,Mem Bsp: mov  ss,old_ss
mov  Mem,Reg      Bsp: mov  Variable,ax
mov  Mem,Seg. Reg Bsp: mov  Var,es
mov  Mem,Wert     Bsp: mov  Var2,12

```

Abk.: Reg=CPU Register, Seg. Reg=Segment-Register, Mem=Memory(Speicher)

MOVS

MOVSB (MOVE String, Byte) bzw. MOVSW (MOVE String, Word) überträgt ein Byte oder Word das durch DS:SI adressiert wird nach ES:DI, MOVS stellt also eine Kombination von LODS und STOS dar. Nach jeder Übertragung werden SI und DI je nach Direction Flag erhöht (default) oder erniedrigt (je nach Direction Flag - siehe auch STD bzw. CLD). Mit REP können mehrere Daten kopiert werden.

Beispiel:

```

lea  si,Quelle    ; Woher kopieren ?
lea  di,Ziel     ; Wohin kopieren (vorher es setzen)
mov  cx,Laenge   ; Wieviel kopieren ?
rep  movsb       ; Kopieren

```

MUL

MUL Multipliziert den vorzeichenlosen Wert des Operanden mit dem vorzeichenlosen Wert im Akkumulator miteinander. Man unterscheidet zwischen Byte und Word Multiplikation.

Byte Multiplikation:

Der erste Wert muss sich im AL Register befinden, der andere wird hinter MUL angegeben und muss sich in einem 8 Bit Register oder einer Byte Speicherstelle befinden. Das Ergebnis wird in AX abgelegt.

Beispiel:

```

mov  bl,10
mov  al,8
mul  bl      ;==> ax = 80

```

```

mov  al,20
mov  Var1,20
mul  Var1    ;==> ax = 400

```

Word Multiplikation:

Ein Operand muss sich im AX Register befinden, der andere in einem 16 Bit Register oder einer Word Speicherstelle. Der High-Teil des Ergebnisses wird in DX abgelegt, der Low-Teil in AX.

Beispiel:

```

mov  ax,350
mul  ax      ;DX:AX = AX * AX

```

[Weiter](#) zu Teil 2.

[Zurück](#) [Weiter](#) [Inhalt](#)



NOP

Null Operation. Ausser IP wird nichts verändert. [90]

NOT

Invertiert alle Bits im Operanden: 0 --> 1 ; 1 --> 0

Beispiel:

```
mov ah,10010100b ;ah=10010100b
not ah           ;ah=01101011b
```

OR

Logische ODER-Verknüpfung: Der Zieloperand wird bitweise mit dem Quelloperand verknüpft. OR Ziel,Quelle. Das Ergebnis wird im Zieloperand abgelegt.

Beispiel:

```
mov ah,10010100b ;ah=10010100b
mov al,10110101b ;al=10110101b
or  ah,al        ;ah=10110101b
```

OUT

Ausgabe eines Bytes oder Words an eine Portadresse: OUT Portadresse,AX bzw. AL.

Wenn die Portadresse grösser als 255 ist muss der Wert in DX stehen sonst kann der als Direktwert angegeben werden.

Beispiel:

```
mov al,'% '
out 60h,al ;Schreibt das Zeichen % in den Tastaturpuffer (Adresse 60h)
```

!POP

Der Befehl POP holt den letztgespeicherten WORD Wert vom Stack und erhöht danach den Stackpointer um den Wert 2.

Beispiel:

```
push ax ;ax sichern
pop ax  ;ax holen
```

POPF

Holt analog zum Befehl POP das Statusregister, d.h. die Flags vom Stack.

Beispiel:

```
pushf ;Flags sichern
popf  ;Flags holen
```

!PUSH

PUSH verringert den Stackpointer um 2 und Speichert ein WORD auf dem Stack.

Beispiel:

```
push es ;es sichern
pop ds  ;Gesicherten Wert nach ds
```

PUSHF

Speichert analog zum Befehl PUSH das Statusregister, d.h. die Flags auf dem Stack.

Beispiel:

```
pushf    ;Flags sichern
popf     ;Flags holen
```

REP

REP wiederholt (REPeat) den nachfolgenden Stringbefehl bis CX gleich null ist. REP decrementiert selbständig. Es existieren folgende Varianten des Befehls: REPE/REPZ wiederhole bis CX = 0 und ZF = 1 ; REPNE/REPNZ wiederhole bis CX = 0 und ZF = 0

Beispiel:

Siehe die einzelnen Stringbefehle

!RET

RET (RETurn) kehrt aus einem Unterprogramm zurück, welches durch einen CALL - Befehl aufgerufen wurde.

Beispiel:

```
.
.
call Write      ;Das Unterprogramm Write aufrufen
.
.
Write PROC Near ;Deklaration des Unterprogramms
mov  ah,09h     ;Beispielcode
int  21h
ret             ;Zurück
Write ENP       ;Ende der Prozedur
[C3;CB;C2;CA]
```

STC

STC setzt das Carry Flag auf 1 (SeT Carry flag).

Beispiel:

```
clc    ;CF = 0
stc    ;CF = 1
```

STD

Setzt das Direction Flag, dadurch verlaufen alle Stringoperationen von rechts nach links zur kleineren Adresse hin.

STI

Mit STI (SeT Interrupt flag) werden die Interrupts die mit CLI gesperrt wurden wieder zugelassen. Siehe auch CLI.

STOS

STOSB überträgt den Inhalt von AL in einen durch ES:DI adressierten Speicherbereich.

STOSW überträgt den Inhalt von AX. Mit REP können mehrere Daten übertragen werden.

SUB

SUB Ziel,Quelle subtrahiert den Inhalt des Quelloperanden vom Zieloperanden. Das Ergebnis wird im Zieloperanden abgelegt.

Beispiel:

```
mov  ah,100      ; ah=100
sub  ah,25       ; ah=75
mov  ah,99       ; ah=99
mov  al,09       ; al=9
sub  ah,al       ; ah=90
```

TEST

TEST verknüpft den Zieloperanden bitweise mit dem Quelloperanden durch eine UND Verknüpfung. Das Ergebnis wird nicht abgespeichert, sondern verändert nur das Statusregister.

Beispiel:

Man will prüfen, ob das Bit 6 von ah gleich eins oder null ist:

```
test ah,01000000b
je   Label1      ;Springe wenn Bit 6 = 1
jne  Label2      ;Springe wenn Bit 6 = 0
```

XCHG

Tauscht den Inhalt eines Registers mit einem anderem Register oder einem Speicherplatz.

Beispiel:

```
mov  ax,70h      ; ax=70h
```

```
mov    bx,40h      ; bx=40h
xchg   ax,bx       ; eXCHanGe: Austauschen der Inhalte ==> ax = 40h  bx = 70h
```

! XOR

Logische XOR-Verknüpfung: Der Zieloperand wird bitweise mit dem Quelloperand verknüpft. XOR Ziel,Quelle. Das Ergebnis wird im Zieloperand abgelegt.

Beispiel:

```
mov    ah,10010100b ;ah=10010100b
mov    al,10110101b ;al=10110101b
xor    ah,al         ;ah=00100001b

xor    ax,ax         ;ax=0 (Schneller als mov ax,00)
```

Weiter geht es mit einem [Programm](#).

[Zurück](#) [Weiter](#) [Inhalt](#)



Das Programm "Wie geht's soll zeigen, wie man dem Benutzer eine Frage stellen kann, auf die er mit ja oder nein antworten kann und dann eine Entsprechende Antwort erhält. In unserem Fall soll es die Frage "Geht es Ihnen gut?" sein. Der Benutzer soll auf diese Frage antworten können, indem er entweder "j" für ja oder "n" für nein drückt. Lautet die Antwort des Benutzers ja dann soll der Computer mit "Toll!" antworten, lautet die Antwort des Benutzers nein soll die Computerantwort "Schade!" lauten. Ich werde im folgenden den Code für eine EXE Datei zeigen, und danach den für eine COM Datei.

Programm für eine EXE Datei:

```

.MODEL SMALL                                ;Standart Speichermodell fuer EXE
.STACK 100h                                ;100h Stack sind mehr als genug
.DATA                                       ;Beginn des Datensegements
Frage DB "Geht es Ihnen gut (j/n)?$"        ;Die Frage und die Antworten enden
Antwort1 DB 13,10,"Toll !$"                ;mit einem $ um DOS zu zeigen, dass
Antwort2 DB 13,10,"Schade !$"              ;es keine Zeichen mehr ausgeben soll
                                           ;13=Carriage Return
                                           ;10=Zeilenvorschub, d.h. zur naechsten
                                           ;Zeile an den Anfang gehen.

.CODE                                       ;Beginn des Codesegementes
mov ax,@data                               ;@data ist eine Vordefinierte Konstante,
                                           ;sie enthaelt die Adresse des
                                           ;Datensegements

mov ds,ax                                  ;ds = ax = @data => Datensegment bereit
mov dx,OFFSET Frage                        ;Die Offsetadresse der Frage nach dx
mov ah,09h                                ;DOS Funktion 9: Stringausgabe.
                                           ;Ausgegeben wir der String an DS:DX
                                           ;bis $
int 21h                                    ;DOS Funktion 9 ausfuehren!

mov ah,07h                                ;DOS Funktion 7: Zeichen von Tastatur
int 21h                                    ;lesen. Das Zeichen wird in al
                                           ;ausgegeben.

cmp al,'j'                                  ;Vergleiche al mit j ACHTUNG SIEHE UNTEN
je Toll                                    ;Wenn gleich Springe zum Label Toll
cmp al,'J'                                  ;Vergleiche al mit J ACHTUNG SIEHE UNTEN
je Toll                                    ;Wenn gleich springe zum Label Toll

NichtToll:                                  ;Dieses Label dient nur der Uebersicht.
mov ah,09h                                ;DOS Funktion 9: Stringausgabe
lea dx,Antwort2                            ;Anstatt mov dx,OFFSET Antwort2
int 21h                                    ;DOS Funktion ausfuehren.
jmp Ende                                    ;Zum Label Ende springen.

Toll:                                       ;Ausgabe der Antwort 1
mov ah,09h
lea dx,Antwort1

```

```

                                int    21h
Ende:
                                mov     ah,4Ch           ;DOS Funktion Programm beenden.
                                int     21h           ;Programm beenden, Kontrolle ans
                                                ;Betriebssystem geben.
                                END                 ;Assembleranweisung: Ende des Programmes

```

--> Erstellen der EXE-Datei mit TASM Dateiname und TLINK Dateiname

Bsp.: Ihre Datei heisst frage.asm, tippen sie also: Tasm frage [Return] und dann Tlink frage[Return].

Programm für eine COM Datei:

```

                                .MODEL TINY           ;Speichermodell fuer COM Dateien
                                .CODE                ;Daten kommen ebenfalls ins Codesegm.
                                ORG 100h             ;Eine COM Datei muss im Speicher bei
                                                ;Offset 100h beginnen, ORG 100h definiert
                                                ;die Adresse des naechsten Befehls
                                                ;==> Mehr dazu spaeter !
                                                ;CS=DS=SS=ES, DS muss nicht initialisiert
                                                ;werden

Start:
                                mov     ah,09h
                                mov     dx,OFFSET Frage
                                int     21h           ;Frage ausgeben
                                mov     ah,07h
                                int     21h           ;Antwort nach al
                                cmp     al,'j'        ;Vergleiche al mit j ACHTUNG SIEHE UNTEN
                                je      Toll          ;Wenn gleich Springe zum Label Toll
                                cmp     al,'J'        ;Vergleiche al mit J ACHTUNG SIEHE UNTEN
                                je      Toll          ;Wenn gleich springe zum Label Toll

NichtToll:
                                mov     ah,09h
                                lea     dx,Antwort2
                                int     21h           ;Antwort 2 ausgeben.
                                jmp     Ende          ;Zum Label Ende springen.

Toll:
                                mov     ah,09h         ;Ausgabe der Antwort 1
                                lea     dx,Antwort1
                                int     21h

Ende:
                                mov     ah,4Ch         ;DOS Funktion Programm beenden.
                                int     21h

```

;Die Daten kommen erst am Ende, da diese sonst als Programmcode ausgefuehrt wuerden (es sei denn, ;man ueberspringt diese mit jmp), dies haette unvorhersehbare folgen. Nach der DOS Funktion 4Ch ;sind die Daten jedoch sicher, da das Programm ja vorher beendet wird bevor CS:IP die Daten ;erreicht.

```

Frage      DB  "Geht es Ihnen gut (j/n)?$"
Antwort1   DB  13,10,"Toll !$"
Antwort2   DB  13,10,"Schade !$"

```

```

                                END Start           ;Definiert das Label Start als Eingangs-
;punkt (Entrypoint) bei COM Dateien kann ich keinen Sinn erkennen (es gehoert so) bei EXE
;Dateien hat man jedoch die Moeglichkeit den Entrypoint nicht nur an den Anfang des Programmes
;zu setzen, sondern auch in der Mitte oder sonstwo. Dies ermoeoglicht der EXE Header.

```

--> Erstellen der COM-Datei mit TASM Dateiname und TLINK /t Dateiname (kleines t !!)

Bsp.: Ihre Datei heisst frage.asm, tippen sie also: Tasm frage [Return] und dann Tlink /t frage [Return].

Wichtiges zu COM-Dateien.

In einer COM Datei müssen Daten und Code in ein Segment passen, daher dürfen COM-Dateien maximal 64 KB gross sein (eine

Ausnahme macht hier Edit.com die Datei ist grösser als 64 KB, sie hat aber einen EXE-Header, und ist daher eigentlich auch eine EXE-Datei). Eine COM-Datei hat aber auch Vorteile: Sie ist kleiner als eine vergleichbare EXE. Wenn Sie eine COM-Datei programmieren, brauchen Sie dennoch nicht auf einen Stack verzichten, er befindet sich im selben Segment wie der Code (am Ende), darf also nicht zu groß werden da er sonst Daten oder Code überschreibt!

Existieren im selben Verzeichnis eine COM und eine EXE Datei mit gleichem Namen, so wird die COM Datei immer zuerst gestartet. Es ist zumindest mir schon passiert, dass ich ein "COM-Projekt" auf "EXE" umgestellt habe und vergessen habe die COM-Datei zu löschen. Anstatt meiner verbesserten "EXE" habe ich dann immer die alte "COM" gestartet und mich gewundert, dass sich nichts tut na ja so weit so gut...

Beim Start einer COM-Datei ist CS=SS=ES=DS.

Noch was: Das Windows 95 – DOS Fenster hat viele Vorteile und nur wenige Nachteile:

Vorteile im Falle eines Fehlers, wenn sich der Computer z.B. aufhängt kann man mit Alt Tab einfach zu Windows wechseln und dann das Problem mit einem Rechtsklick auf den Programmknopf in der Task-Leiste beseitigen. Auch die Tatsache dass man auf einer virtuellen Maschine arbeitet bringt Vorteile, da diese eine eigene Interruptvektortabelle hat. Hat man diese durch speicherresidente Programme verändert, dann kann man einfach mit exit die virtuelle Maschine beenden und das todbringende Speicherresidente Programm kann auch keinen Schaden mehr anrichten. Der Nachteil ist, dass bestimmte Programme im DOS-Fenster gar nicht laufen, sondern nur im "echten" DOS. Das ist mir auch schon passiert, kommt aber nur sehr selten vor.

ACHTUNG:

Windows-Editoren verwenden einen anderen Zeichensatz als DOS-Editoren und der Assembler. Sie sollten daher immer ihre Programme mit dem DOS-Editor schreiben, da sonst der Assembler Fehler melden kann, weil er den Code nicht "versteht". Wenn Sie also die Programme nicht abschreiben, sondern unter Windows kopieren, sollten Sie vorher den Code unbedingt mit dem DOS-Editor überprüfen. Anmerkung: Seit geraumer Zeit verwende ich EditPlus 1.25 für die Programmierung und habe keinerlei Probleme damit! Sieh Anhang!

Weiter geht es mit einem weiteren [Programm](#).

[Zurück Weiter Inhalt](#)



Mit diesem kleinen Programm will ich zeigen, wie man den Status der Num–Lock Taste verändern kann. Auch möchte ich zeigen, wie man eine beliebige Stelle im Speicher manipulieren kann. Bei der physikalischen Adresse 0417h befindet sich das Statusbyte des Keyboards. Bit 5 ist für Num–Lock zuständig. Ist es gesetzt, ist Num–Lock an und die Signallampe auf der Tastatur leuchtet.

Programm als COM-Datei:

```
.MODEL TINY                                ;Speichermodell fuer COM Dateien
.CODE
ORG 100h

Start:
    mov ax,0040h
    mov es,ax                               ;Segmentadresse = 40h (Segmentoverwrite)
    xor byte ptr ES:[17h],00100000b        ;Bit 5 "umpolen"
    mov ah,4Ch
    int 21h                                 ;Programm beenden.
    END Start
```

Gleiches Programm etwas anders (Adressierung etw. anders):

```
.MODEL TINY                                ;Speichermodell fuer COM Dateien
.CODE
ORG 100h

Start:
    mov ax,0000h
    mov es,ax                               ;Segmentadresse = 0h (Segmentoverwrite)
    xor byte ptr ES:[0417h],00100000b      ;Bit 5 "umpolen".
    mov ah,4Ch
    int 21h                                 ;Programm beenden.
    END Start
```

Info:

Was bedeutet xor byte ptr ES:[17h] ? Byte ptr sagt dem Assembler, das er ein Byte bearbeiten soll, WORD PTR würde ein Word bearbeiten. Beispiel: Man möchte an die Adresse 3000 den Wert 40h speichern. Mit mov [3000],40h weiss der Assembler noch nicht, ob er ein Word oder ein Byte kopieren soll. Mit Byte / Word PTR muss man also die Breite angeben: mov byte ptr [3000],40h. Die eckige Klammer signalisiert, dass der Inhalt einer Speicherstelle gemeint ist. ES:[17h] zeigt den genauen Speicherort an, welcher aus Segmentadresse und Offset- adresse besteht. Man könnte anstatt des Extrasegmentes hier auch das Datensegment zur Adressierung verwenden.

Info's zur indirekten/indizierten Adressierung:

Bei der indirekten Registeradressierung wird die Offsetadresse durch ein oder mehrere der folgenden Register gebildet: BX, BP, DI, und SI. Es gibt folgende Kombinationsmöglichkeiten: [bx] ; [bp] ; [di] ; [si] ; [bx+di] ; [bx+si] ; [bp+di] ; [bp+si] .

Bei der indizierten Adressierung kommt zu den Registern noch ein konstanter Wert hinzu (z.B. der eines Labels). Es gibt dann folgende Kombinationen:

[Label+bx] ; [Label+bp] ; [Label+di] ; [Label+si] ; [Label+bx+di] ; [Label+bx+si] ;
[Label+bp+di] ; [Label+bp+si] .

Diese beiden Adressierungsarten haben den Vorteil, dass der Wert veränderbar ist und erst während des Programmlaufes festgelegt wird. Man kann damit zum Beispiel auf Tabellen zugreifen und so weiter. Standardmäßig wird als Segmentregister das Datensegment verwendet ist bp beteiligt wird das Stacksegment verwendet. Diese beiden Adressierungsarten sind für den Anfang jedoch noch nicht so wichtig.

Weiter geht es mit einem weiteren [Programm](#).

[Zurück](#) [Weiter](#) [Inhalt](#)



Ich werde zwei Programme vorstellen, mit denen man den Inhalt des CMOS-RAM sichern und danach bei Bedarf wieder herstellen kann.

Diese Programme zeigen, wie man Daten zu einem Port schickt bzw von diesem erhält. Das Programm zu wiederherstellen habe ich getestet, (auf einem P90 mit AMIBIOS) und es hat funktioniert, aber ist es trotzdem möglich, dass es unter bestimmten Umständen Schaden anrichten kann. In diesem Fall: Ich habe Sie gewarnt. Das Sicherungsprogramm funktioniert. Mehr Informationen zum CMOS-RAM finden sie in dieser [Textdatei](#).

Vorbemerkungen zur Datei-Verwaltung mit Handles:

Bevor man etwas in eine Datei schreiben kann, oder etwas aus ihr lesen muss man sie öffnen. Dies geschieht mit der DOS-Funktion 3Dh. AH muss also mit 3Dh belegt werden, in AL kommt der Zugriffsmodus: 0 für nur Lesen, 1 für nur Schreiben und 2 für Lesen und schreiben. Der Name der Datei wird über DS:DX angegeben. Wenn das Öffnen klappt, bekommt man in AX ein sogenanntes Handle. Öffnet man eine weitere Datei, dann erhält man ein anderes Handle. Beim Zugriff auf eine Datei muss immer das Handle angegeben werden, es dient also zur Identifikation der Datei(en). Folgende Handles sind vorbesetzt, man kann mit ihnen und mit der Hilfe der DOS-Funktionen auch auf Ausgabegeräte schreiben/lesen, diese also wie eine Datei behandeln:

<u>Handle:</u>	<u>Gerät:</u>	<u>Bezeichnung:</u>	
1	Bildschirm	Standart-Ausgabe	(STDOUT, CON)
2	Bildschirm	Standart-Fehlerausgabe	(STDERR)
3	Serielle Schnittstelle	Hilfsein-/ausgabe	(STDAUX, AUX, COM)
4	Drucker/Parallele Schnittst.	Standart-Drucker	(STDPRN, PRN, LPT)

Wenn man die Datei erst erstellen muss, dann benutzt man DOS-Funktion 3Ch. Man erhält dann ebenfalls eine Handle in AX. Der Dateiname muss über DS:DX angegeben werden. Für beide Fälle (öffnen/erstellen) gilt: Der Dateiname muss so: Lbl DB "Dateiname.ext",0 beendet werden also durch das ASCII Zeichen 00h. Mit der Funktion 3Fh kann man dann aus der Datei (bzw. dem Gerät) lesen, mit Funktion 40h kann man schreiben. Das Handle muss diesen Funktionen immer in BX übergeben werden, die Anzahl der zu lesenden/schreibenden Bytes muss in CX übergeben werden. Die gelesenden/zu schreibenden Daten kommen/steht in den/im Puffer, der über DS:DX adressiert wird. Wenn man Fertig ist, muss die Datei mit Funktion 3Eh geschlossen werden, das Handle muss wieder in BX stehen. Bei allen Handle-Funktionen gilt: Wird nach der Funktion das Carry-Flag gesetzt, dann ist ein Fehler aufgetreten. Dies kann mit JC überprüft werden.

Zusammenfassung:

1. Öffnen einer Datei:

Interrupt 21h Funktion 3Dh.

Eingabe:

AH = 3Dh

AL = Zugriffsmodus (00h = Nur lesen, 01h = Nur schreiben, 02h = Lesen und schreiben)

DS = Segmentadresse des Dateinamens

DX = Offsetadresse des Dateinamens

Ausgabe:

Carry Flag = 0 --> Ok

AX = Handle der Datei

Ist ein Fehler aufgetreten dann Carry Flag = 1 und Fehlercode in AX:

AX = 1 = Datei schon geöffnet

AX = 2 = Datei nicht gefunden

AX = 3 = Pfad nicht gefunden
AX = 4 = Kein freies Handle
AX = 5 = Zugriff verweigert

2. Datei erstellen:

Interrupt 21 Funktion 3Ch

Eingabe:

AH = 3Ch

DS = Segmentadresse des Dateinamens

DX = Offsetadresse des Dateinamens

CX = Dateiattribut (00h = Normal, 01h = Schreibschutz, 02h = Versteckt, 04h = System)

Ausgabe:

Carry Flag = 0 --> OK

AX = Handle der Datei -->D.h. die Datei braucht nicht noch extra geöffnet werden!

Ist ein Fehler aufgetreten dann Carry Flag = 1 und Fehlercode in AX:

AX = 1 = Pfad nicht gefunden

AX = 4 = Kein freies Handle

AX = 5 = Zugriff verweigert

--> Falls die angegebene Datei bereits existiert, wird sie ohne Warnung überschrieben.

3. Datei schliessen:

Interrupt 21 Funktion 3Eh

Eingabe:

AH = 3Eh

BX = Handle der Datei

Ausgabe:

Carry Flag = 0 --> OK

Carry Flag = 1 --> Fehler u. Fehlercode in AX (AX = 6 = Ungültiges Handle)

4. Datei lesen:

Interrupt 21h Funktion 3Fh

Eingabe:

AH = 3Fh

BX = Handle der Datei/des Geräts

CX = Anzahl der zu lesenden Bytes

DS = Segmentadresse des Puffers

DX = Offsetadresse des Puffers

Ausgabe:

Carry Flag = 0 --> OK und AX = Anzahl der gelesenen Bytes

Carry Flag = 1 --> Fehler u. Fehlercode in AX:

AX = 5 = Zugriff verweigert

AX = 6 = Ungültiges Handle

--> Der "Zeiger" innerhalb der zu lesenden Datei wird automatisch um den Wert in CX erhöht, so dass der nächste Lesevorgang wieder neue Daten einliest.

5. Datei schreiben:

Interrupt 21h Funktion 40h

Eingabe:

AH = 40h

BX = Handle der Datei/des Geräts

CX = Anzahl der zu schreibenden Bytes

DS = Segmentadresse des Puffers

DX = Offsetadresse des Puffers

Ausgabe:

Carry Flag = 0 --> Ok und AX = Anzahl der geschriebenen Bytes

Carry Flag = 1 --> Fehler u. Fehlercode in AX:

AX = 5 = Zugriff verweigert

AX = 6 = Ungültiges Handle

--> Auch bei dieser Funktion wird der Zeiger innerhalb der Datei automatisch um den Wert in CX erhöht, so dass der nächste Schreibvorgang hinter dem zuletzt geschriebenen Byte erfolgt.

Und hier das Programm zum sichern der CMOS-RAM Daten:

```
.MODEL SMALL  
.STACK 100h  
.DATA
```

```

Dateinamen      DB    "CMOS.DAT",0    ;In diese Datei kommen die Daten. Die 0 zeigt das Ende an
Puffer          DB    ?
FehlerMeld      DB    "Fehler beim erstellen/schreiben/schliessen der Datei CMOS.DAT !$"
Msg1            DB    "CMOS RAM Sicherungsprogramm Version 1.03",13,10,"$"

```

```

        .CODE
cli                ;extern maskierbare Interrupts sperren
mov  ax,@data
mov  ds,ax
mov  dx,OFFSET Msg1
mov  ah,09h
int  21h
mov  dx,OFFSET Dateinamen
xor  cx,cx        ;cx=0 -->Dateiattribut Normal.
mov  ah,3Ch        ;DOS Funktion: Create File with Handle
int  21h          ;Datei Erstellen. Ist eine gleichnamige Datei bereits
;vorhanden, dann wird diese ueberschrieben.
;*****
jc   Fehler       ;Springe wenn Fehler
xchg bx,ax        ;Handle sichern. Das Handle dient als eine Art Kennung der
;Datei, wenn auf diese zugegriffen wird.
;*****
mov  cx,80h        ;CMOS Maximaladresse in cx
;Das CMOS-RAM ist bei neueren Computern 128 Bytes gross (80h)
xor  ah,ah        ;ah Zeigt auf tatsaechliche CMOS-RAM Adresse

Schleife:
mov  al,ah
out  70h,al        ;In diesen Port kommt die Offsetadresse im CMOS-RAM rein
jmp  $+2          ;Der Befehl soll nur Zeit verbrauchen.
;Das $ wird hier vom Assembler als die aktuelle Position
;im Code interpretiert. Gesprungen wird also direkt zum
;naechsten Befehl (in al,71h)
in   al,71h        ;CMOS Daten aus Port einlesen aus oben eingegebener Adresse.
mov  Puffer,al    ;In den Puffer!
;*****
push cx           ;Schleifenzaehler cx auf Stack
push ax          ;Addresszaehler ax bzw ah auf Stack
mov  ah,40h       ;DOS-Funktion: Write file or Device
mov  cx,01h       ;Laenge der zu Speichernden Daten
mov  dx,OFFSET Puffer ;Puffer von dem Geschrieben wird
int  21h          ;Puffer in Datei Schreiben
jc   Fehler
pop  ax           ;ax vom Stack
pop  cx           ;cx vom Stack
inc  ah           ;ah Erhoehen!
Loop Schleife
;*****
mov  ah,3Eh       ;Datei schliessen
int  21h
jnc  Ende
mov  dx,OFFSET FehlerMeld
mov  ah,09h
int  21h          ;Fehlerbehandlung fuer Arme
;*****

Ende:
sti                ;Extern maskierbare Interrupts zulassen
mov  ah,4Ch
int  21h          ;Und Tschuess
END

```

;-->Dieses Programm speichert also die CMOS-RAM Daten in der Datei Cmos-dat, welche ins selbe
 ; Verzeichnis kommt.

Und hier das Programm zum wiederherstellen der CMOS-RAM Daten:

```

.MODEL SMALL
.STACK 100h
.DATA

```

```
Dateiname      DB      "CMOS.DAT",0
Puffer         DB      ?
FehlerMeld     DB      10,13,"Fehler beim oeffnen/lesen/schliessen der Datei!"
               DB      "Datei CMOS.DAT vorhanden?$"      ; Zur Wiederherstellung muss sich die Datei Cmos
;mit dem Sicherungsprogramm erstellt)
```

Schlussbemerkung:

Diese beiden Programme sind sicher nicht der Weisheit letzter Schluss, so könnte man z.B. beide Programme vereinen, die Fehlerbehandlung verbessern etc. Ich hab sie trotzdem gebracht, weil sie halbwegs kurz sind, wenn Sie ein besseres Programm wollen... nur zu!

Ist mir gerade eingefallen: Besser ist es wohl wenn man die Daten zuerst in eine 128 Byte Variable speichert und die CMOS-Daten erst am Schluss in die Datei bzw. ins CMOS-RAM schreibt das wäre schneller und auch einfacher, byteweise geht es allerdings auch. Für Moderne PC's ist der Gebrauch nicht mehr sinnvoll da mehr als 128 Bytes im RAM gespeichert sind!

Weiter geht es mit einem weiteren [Programm](#).

[Zurück Weiter Inhalt](#)



Die Idee für diese Programm ist mir gekommen, als ich an diese Homepage gearbeitet hab. In HTML sollte man die Umlaute ja bekanntlich besser "Maskieren". Wenn man nun seine Homepage mit dem mächtigen Notepad-Tool erstellt darf man wegen jedem noch so dummen ä ein ä schreiben, das Gleiche gilt für ü,ö usw. Was liegt nun also näher, als ein Konvertierungsprogramm zu schreiben, das alle ü's und ä's usw. durch ü's und ä's usw. ersetzt. Das Programm wird über sogenannte Kommandozeilenparameter gesteuert, d.h. der Pfad und der Dateiname wird nach dem eigentlichen Programmnamen eingegeben:

Konvert [LW:][Pfad\]Dateiname.ext . Alles was in [] steht ist wieder optional. Das Programm kopiert dann die Datei Byte für Byte [ziemlich langsam, aber für Internetprojekte durchaus schnell genug, da diese meist (was den Text angeht)<< 1 MB sind] und ersetzt gegebenenfalls Zeichen durch Textstrings. Die so konvertierte Datei wird mit gleicher Erweiterung aber mit leicht verändertem Dateinamen abgespeichert und muss nun nur noch ggf. umbenannt werden. Die Kommandozeile wird dem Programm über das sogenannte PSP (Programm Segment Präfix) übergeben. Das PSP ist 256 Bytes gross (100h) und beinhaltet diverse Informationen, bei COM- Dateien befindet es sich immer direkt davor deshalb das ORG 100h ! Beim Programmstart zeigen bei einer COM- und EXE-Dateien ES und DS auf den Anfang des PSP's oder genauer ES:0000 bzw. DS:0000. Die Kommandozeile, also der Textstring den der User nach dem eigentlichen Namen der EXE- bzw. COM-Datei eingibt befindet sich, inklusive des Leerzeichens zwischen dem Dateinamen und der Kommandozeile, im PSP ab der Adresse 81h, d.h. da man das Leerzeichen meist nicht will ab der Adresse 82h. Die Länge der Kommandozeile (auch wieder plus Leerzeichen) befindet sich in dem Byte bei 80h. Zuerst werden die Kommandozeilen-Daten in einer Variablen im Datensegment mit rep movsb abgespeichert. Diese Variable ist mit 0 (Ascii 00h) vorbelegt, so dass am Ende auf jeden Fall eine 0 steht.

Aufbau des Programms:

1. Die Kommandozeile wird in der Variable CommandTail gespeichert.
2. Die Eingabe wird rudimentär auf Fehler überprüft:
Wenn die Kommandozeile kleiner oder gleich 4 Zeichen ist wird zu NoTail gesprungen, die Fehlermeldung Msg2 ausgegeben und das Programm beendet.
3. Der Textstring mit dem Label Message wird ausgegeben.
4. Die durch die Kommandozeile spezifizierte Datei wird im "nur lesen Modus" geöffnet.
Wenn dies nicht klappt, wird zum Label Ende gesprungen, die Fehlermeldung Msg3 ausgegeben und das Programm beendet.
5. Das Handle der vorher geöffneten Datei wird in der Variable Datei1 gespeichert.
6. Es wird eine zweite Datei erstellt (da kommt der konvertierte Text rein). Der Name dieser Datei wird aus dem Namen der ersten Datei erstellt, welcher allerdings vorher wie folgt verändert wird: Das Zeichen an der 5. Stelle von hinten (das ist das Zeichen vor dem Punkt) wird durch eine "0" ersetzt oder falls er da schon eine "0" hat dich ein "X". Dazu nimmt man die Startadresse von CommandTail in bx und die Länge in di. Von di wird nun 6 abgezogen (5 Zeichen zurück + 1 Leerzeichen). Nun kann man das Zeichen über die indirekte Registeradressierung (hier: [bx+di]) überprüfen und verändern.
7. Das Handle der neuen Datei wird in der Variable Datei2 gespeichert.
8. Aus der ersten Datei wird ein Byte gelesen.
9. In mehreren aufeinanderfolgenden Schritten wird nun das Byte überprüft und ggf. ersetzt.
Wenn es kein Umlaut ist, wird das Originalbyte geschrieben.
-> Schritt 8 und 9 werden so lange wiederholt, bis die Datei vollständig konvertiert wurde.
(Siehe Code!)

--> Das Programm funktioniert nur bei Texten, die mit einem Windows (Ansi) Editor gemacht wurden. Um es für DOS umzubauen, muss man nur die Ansi-Zahlen durch DOS Zeichen ersetzen ! Also z.B.: statt 252 schreibt man 'ü' !

Und hier das Programm als EXE:

```

.MODEL SMALL
.STACK 100h
.DATA
CommandTail DB 126 DUP (0)
Puffer1 DB ?
Datei1 DW ?
Datei2 DW ?
KleinUE DB "&uuml;"
GrossUE DB "&Uuml;"
KleinOE DB "&ouml;"
GrossOE DB "&Ouml;"
KleinAE DB "&auml;"
GrossAE DB "&Auml;"
CountIt DW ?
Message DB "HTML Textkonverter WIN Ver 1.00 (c) 1998 by Marcus Roming$"
Msg2 DB "Kein oder falscher Parameter. Syntax: Winkonv [LW:\][Path\]FileName."
DB "Ext $" ; Ende! (ganz raus aus der Schleife)
jmp Fertig

Ziel3:
cmp byte ptr [Puffer1],252 ;Ansi-Zahl fuer ü
jne Weiter1 ;Wenn nicht gleich auf den naechsten Pruefen
mov ah,40h ;Wenn gleich die Daten aus KleiUE schreiben
mov cx,06h
lea dx,KleinUE
mov bx,[Datei2]
int 21h
jmp Raus ;Es wurde ein ü gefunden, es kann also kein
;anderes mehr kommen--> Sprung ans Ende der
;Schleife (nicht ganz raus !!))

Weiter1:
cmp byte ptr [Puffer1],220 ;Ansi-Zahl fuer Ü
jne Weiter2
mov ah,40h
mov cx,06h
lea dx,GrossUE
mov bx,[Datei2]
int 21h
jmp Raus

Weiter2:
cmp byte ptr [Puffer1],228
jne Weiter3
mov ah,40h
mov cx,06h
lea dx,KleinAE
mov bx,[Datei2]
int 21h
jmp Raus

Weiter3:
cmp byte ptr [Puffer1],196
jne Weiter4
mov ah,40h
mov cx,06h
lea dx,GrossAE
mov bx,[Datei2]
int 21h
jmp Raus

Weiter4:

```

```

        cmp     byte ptr [Puffer1],246
        jne     Weiter5
        mov     ah,40h
        mov     cx,06h
        lea     dx,KleinOE
        mov     bx,[Datei2]
        int     21h
        jmp     Raus

Weiter5:
        cmp     byte ptr [Puffer1],214
        jne     Weiter6
        mov     ah,40h
        mov     cx,06h
        lea     dx,GrossOE
        mov     bx,[Datei2]
        int     21h
        jmp     Raus

Weiter6:
        mov     bx,[Datei2]
        mov     ah,40h
        mov     cx,01h
        lea     dx,Puffer1
        int     21h

Raus:
        jmp     DoLoop           ;Unbedingter Sprung nach DoLoop

Fertig:
        mov     bx,[Datei1]
        mov     ah,3Eh
        int     21h

        mov     bx,[Datei2]
        mov     ah,3Eh
        int     21h
        jmp     GanzFertig

NoTail:
        mov     ax,@data         ;Noch schnell nachholen
        mov     ds,ax
        lea     dx,Msg2
        mov     ah,09h
        int     21h             ;Fehlermeldung ausgeben
        jmp     GanzFertig

Ende:
        lea     dx,Msg3
        mov     ah,09h
        int     21h             ;Fehlermeldung ausgeben

GanzFertig:
        mov     ah,4Ch
        int     21h

        END

```

--> Syntax Beispiel: Winkonv new.txt
 Beispiel 2 : Winkonv d:\temp\win.htm

Download:

Konverter für Windows (ASM): [Winkonv ASM](#)

Konverter für Windows (EXE): [Winkonv EXE](#)

Nun werde ich die bisher verwendeten [Interruptfunktionen](#) noch einmal auflisten.

[Zurück](#) [Weiter](#) [Inhalt](#)



Hier werde ich die Interruptfunktionen die wir bisher gebraucht haben noch einmal auflisten und noch einige neue.

Bisherige Funktionen:

1. Schreibe Zeichenkette:

Interrupt 21h Funktion 9

Eingabe:

AH = 9

DS = Segmentadresse der Zeichenkette

DX = Offsetadresse der Zeichenkette

--> Ende der Zeichenkette muss durch ein \$ signalisiert werden.

2. Zeichen von der Tastatur lesen:

Interrupt 21h Funktion 7

Eingabe:

AH = 7

Ausgabe:

AL = ASCII Code des Zeichens

Wenn AL = 0 ist, dann wurde ein erweiterter Tastaturcode eingelesen, Funktion nochmal aufrufen, um eigentlichen Code zu lesen.

--> Das Zeichen wird nicht angezeigt, die Funktion wartet auf die Eingabe.

3. Zeichen von der Tastatur lesen:

Interrupt 21h Funktion 8

Analog zu Funktion 7 kann jedoch durch STRG+C abgebrochen werden.

4. Öffnen einer Datei:

Interrupt 21h Funktion 3Dh.

Eingabe:

AH = 3Dh

AL = Zugriffsmodus (00h = Nur lesen, 01h = Nur schreiben, 02h = Lesen und schreiben)

DS = Segmentadresse des Dateinamens

DX = Offsetadresse des Dateinamens

Ausgabe:

Carry Flag = 0 --> Ok

AX = Handle der Datei

Ist ein Fehler aufgetreten dann Carry Flag = 1 und Fehlercode in AX:

AX = 1 = Datei schon geöffnet

AX = 2 = Datei nicht gefunden

AX = 3 = Pfad nicht gefunden

AX = 4 = Kein freies Handle

AX = 5 = Zugriff verweigert

5. Datei erstellen:

Interrupt 21 Funktion 3Ch

Eingabe:

AH = 3Ch

DS = Segmentadresse des Dateinamens

DX = Offsetadresse des Dateinamens

CX = Dateiattribut (00h = Normal, 01h = Schreibschutz, 02h = Versteckt, 04h = System)
Ausgabe:
Carry Flag = 0 --> OK
AX = Handle der Datei -->D.h. die Datei braucht nicht noch extra geöffnet werden!
Ist ein Fehler aufgetreten dann Carry Flag = 1 und Fehlercode in AX:
AX = 1 = Pfad nicht gefunden
AX = 4 = Kein freies Handle
AX = 5 = Zugriff verweigert
--> Falls die angegebene Datei bereits existiert, wird sie ohne Warnung überschrieben.

6. Datei schliessen:

Interrupt 21h Funktion 3Eh
Eingabe:
AH = 3Eh
BX = Handle der Datei
Ausgabe:
Carry Flag = 0 --> OK
Carry Flag = 1 --> Fehler u. Fehlercode in AX (AX = 6 = Ungültiges Handle)

7. Datei lesen:

Interrupt 21h Funktion 3Fh
Eingabe:
AH = 3Fh
BX = Handle der Datei/des Geräts
CX = Anzahl der zu lesenden Bytes
DS = Segmentadresse des Puffers
DX = Offsetadresse des Puffers
Ausgabe:
Carry Flag = 0 --> OK und AX = Anzahl der gelesenen Bytes
Carry Flag = 1 --> Fehler u. Fehlercode in AX:
AX = 5 = Zugriff verweigert
AX = 6 = Ungültiges Handle
--> Der "Zeiger" innerhalb der zu lesenden Datei wird automatisch um den Wert in CX erhöht, so dass der nächste Lesevorgang wieder neue Daten einliest.

8. Datei schreiben:

Interrupt 21h Funktion 40h
Eingabe:
AH = 40h
BX = Handle der Datei/des Geräts
CX = Anzahl der zu schreibenden Bytes
DS = Segmentadresse des Puffers
DX = Offsetadresse des Puffers
Ausgabe:
Carry Flag = 0 --> Ok und AX = Anzahl der geschriebenen Bytes
Carry Flag = 1 --> Fehler u. Fehlercode in AX:
AX = 5 = Zugriff verweigert
AX = 6 = Ungültiges Handle
--> Auch bei dieser Funktion wird der Zeiger innerhalb der Datei automatisch um den Wert in CX erhöht, so dass der nächste Schreibvorgang hinter dem zuletzt geschriebenen Byte erfolgt.

9. Programm beenden:

Interrupt 21h Funktion 4Ch
Eingabe:
AH = 4Ch
AL = Errorlevel (Optional)

Neue Funktionen:

A. Zeichenkette (String) von der Tastatur lesen:

Interrupt 21h Funktion 0Ah
Eingabe:
AH = 0Ah
DS = Segmentadresse des Puffers
DX = Offsetadresse des Puffers
--> In das erste Byte des Puffers muss die Maximale Anzahl von Zeichen, die max. eingelesen werden sollen. In das zweite Byte des Puffers

wird die Anzahl der tatsächlich eingelesenen Zeichen ausgegeben.
--> Der Puffer sollte so aussehen:

```
Puffer  DB XXh          ;XX=max. Zeichenanzahl XX <255.  
        DB ?           ;Eingelesene Zeichen  
String  DB XX DUP (?) ;Eingegebener String
```

--> Hier gibt es ein **NEUES (!)** [Beispiel](#) dazu.

B. Booten des Systems:

Interrupt 19h Funktion--
Eingabe: Einfach nur INT 19h
--> Funktioniert nur unter True Dos

C. Move Filepointer:

Interrupt 21h Funktion 42h
Eingabe:
AH = 42h
AL = Position relativ zu
 0 -> Dateianfang
 1 -> Aktueller Dateiposition
 2 -> Dateiende
BX = Handle der Datei
CX = High-Word der Position
DX = Low-Word der Position
Ausgabe:
AX = Low-Word der neuen Position
DX = High-Word der neuen Position
--> Positioniert den Zeiger innerhalb einer Datei für die nächste Schreib- oder Leseaktion.

D. Keyboard - Check for keystroke:

Interrupt 16h Funktion 01h
Eingabe:
AH = 01h
Ausgabe:
Gesetztes Zero-Flag wenn keine Taste gedrückt wurde (Funktion sieht im Tastaturpuffer nach)
Wenn Taste gedrückt wurde: Zero-Flag clear und ASCII Zeichen in AL.
--> Gut um z.B. eine Schleife durch Tastendruck zu unterbrechen. Wie z.B. Turbo-Pascal:
 "Repeat until keypressed"

E. Videomodus setzen:

Interrupt 10h Funktion 00h
Eingabe:
AH = 00h
AL = Videomodus (davon gibt es ziemlich viele, die meisten sind aber kartenspez., viele sind einfach schlecht, deshalb nenn ich nur den Besten: 13h - VGA 320x200 256 Farben. Normaler Modus ist 80x25 d.h. Modus 2 oder 3(Farbig).)
Ausgabe:
Keine.

F. Schreibe Grafikpunkt:

Interrupt 10h Funktion 0Ch
--> Vorher mit Fkt. 00h grafikmodus setzen.
Eingabe:
AH = 0Ch
CX = Spaltenkoordinate
DX = Zeilenkoordinate
AL = Farbattribut
Ausgabe: Keine

[Beispiel](#) von mir zu den letzten genannten Interrupts (einfach, aber schlecht kommentiert)

Wird Fortgesetzt...

MEHR: Siehe DOS-Technical-Reference / Ralph Browns Interruptliste



Dieses Programm ersetzt das alte "free" Programm (das wohl nirgendwo mehr funktioniert) und soll vor allem zeigen wie man mit großen Dateien umgeht. Dabei soll von einer Datei die CRC16 errechnet werden. Von der Datei wird zunächst ein Stück in den vorher reservierten Speicher eingelesen und dann davon die CRC berechnet. Dann wird abermals ein Stück eingelesen, wobei die CRC mit dem vorherigen Wert weiterarbeitet. So werden so lange Daten eingelesen, bis in AX durch eine 0 signalisiert wird, daß wir alles eingelesen haben. Da mehrfach eingelesen wird, ist die Dateigröße nicht auf 64 kByte begrenzt, sondern kann mehrere MByte betragen. Das Programm benutzt Prozeduren und Makros aus meinem Buch im MITP Verlag und eine CRC16-Prozedur aus dem Buch von Günter Born (eine CD).
Syntax: CRC16 filename.ext .

Schon eine kleine Veränderung der Datei an einer beliebigen Stelle, bewirkt eine ganz andere CRC.
Das Programm benötigt die Datei macro.inc welche unten als Download bereitsteht!

Programm für eine EXE Datei:

```
.MODEL SMALL
.386
.STACK 100h
.DATA
Filename DB 126 DUP (0) ;Kommandozeile
Segment2 DW ?
Puffer1 DB 6 DUP (?) ;Puffer für das Resultat

;*****ANFANG CODE *****

.CODE
include macro.inc
CutMem
Gettail Filename,Fehler ;Kommandozeile mit Dateiname
call InitData1
mov al,00h
mov ah,3Dh
mov dx,OFFSET Filename
int 21h ;Datei oeffnen
mov bx,ax
jc Fehler

AllocMem 0FFFh
mov [Segment2],ax ;Speicher res.

call InitFarData
xor dx,dx

LoopIt:
push dx
mov ah,3Fh
mov cx,0EFFFh
mov dx,0
int 21h ;Daten einlesen
```

```

        jc    Fehler

        pop    dx

        cmp    ax,0                ;Fertig mit den Daten
        je     Raus                ;bzw. der Datei?

        mov    cx,ax
        mov    si,0

        call   CRC16                ;CRC ausrechnen
        jmp    LoopIt              ;Nächste Dateiportion!

Raus:
        mov    ah,3Eh
        int    21h

        push   ds
        pop    es
        mov    ah,49h
        int    21h                ;Speicher wieder freigeben

        call   InitData1
        push   ds
        pop    es
        mov    ax,dx
        call   Ausgabe

Fehler:
        mov    ah,4Ch
        int    21h                ;Ende Programm

InitData1    PROC NEAR
        push   ax
        mov    ax,@data
        mov    ds,ax
        ;ASSUME DS:@data
        pop    ax
        ret
InitData1    ENDP

InitFardata  PROC NEAR
        push   ax
        mov    ax,[Segment2]      ;DS auf Segment 2
        mov    ds,ax
        pop    ax
        ret
InitFardata  ENDP

; -->Aus Buch: Assembler Programmierung - Günter Born:
; Pass      - DS:SI = pointer to the buffer
;           - CX    = length of the buffer
; Returns   - DX    = CRC16 of the buffer

CRC16    PROC NEAR

        PUSH   AX
        PUSH   BX
        PUSHF
        CLD                                ; Move forward through the buffer

        ;SUB    DX,DX                    ; CRC := 0000h ; von mir auskommentiert

C1:      LODSB                            ; AL := byte at DS:SI
        SUB    AH,AH

```



```

        XCHG AH,AL          ; AX := 256 * AL
        XOR  DX,AX          ; CRC := CRC xor AX

        PUSH CX
        MOV  CX,8

C2:     MOV  BX,DX
        SHL  DX,1

        AND  BX,8000h
        JZ   C3

        XOR  DX,1021h

C3:     LOOP C2
        POP  CX

        LOOP C1

        POPF
        POP  BX
        POP  AX
        RET
CRC16   ENDP

;--> Aus Buch: Assembler - Grundlagen der Programmierung, Roming, Rhode:
Ausgabe PROC NEAR          ;Ausgabeprozedur
        pusha              ;Register sichern
        xor  cx,cx          ;CX=0=Zeichenzähler
        mov  di,OFFSET Puffer1+6 ;DI an Pufferende
        mov  bx,10          ;Nachher durch 10 dividieren

Loop1:   xor  dx,dx          ;DX=0
        div  bx              ;AX/10, Rest in DX bzw. DL
        add  dl,30h          ;DL nach ASCII
        dec  di              ;DI-1
        mov  [di],dl         ;ASCII Ziffer in Puffer
        inc  cx              ;CX+1
        cmp  ax,0            ;Ist AX=0 ?
        jne  Loop1          ;Springe wenn nein

        mov  ah,40h          ;Funktionsnummer
        mov  bx,1            ;Handle-Nummer
        mov  dx,di           ;DS:DX auf String!
        int  21h             ;ASCII-Zahl ausgeben!
        popa                ;Register wiederherstellen
        ret                  ;und zurück!
Ausgabe ENDP                ;Ende Prozedur

END

```

--> Erstellen der EXE-Datei mit TASM Dateiname und TLINK Dateiname
 Bsp.: Ihre Datei heisst CRC16.asm, tippen sie also: Tasm CRC16 [Return] und dann tlink CRC16 [Return].

Download: [Programm inclusive Macro.inc](#)
[Weiter geht es hier!](#)

[Zurück Weiter Inhalt](#)



Hier möchte ich noch ein paar wichtige Dinge ansprechen:

1. Die "alten" Segmentanweisungen:

Die Segmente wird wie folgt Deklariert:

```
DATEN SEGMENT                ;Der Name (hier Daten) kann bel. gewaehlt werden
;Hier kommen die Daten rein
DATEN ENDS ; END Segment
```

```
STAPEL SEGMENT
dw 128 DUP (?) ;Def. der Stackgroesse
STAPEL ENDS
```

```
CODE SEGMENT
ASSUME CS:CODE, DS:DATEN, ES:NOTHING, SS:STAPEL ;<-- hier erfolgt die genaue Zuweisung zu den
; Segmentregistern
```

```
START:
;Hier kommt der Code rein
```

```
CODE ENDS
END START
```

2. Fehlersuche mit dem Debugger:

Um Fehler in einem Programm zu finden benutzt man am besten einen Debugger (z.B Borland Turbo Debugger DOS). Dort kann man das Programm schrittweise abarbeiten (z.B. mit F7), die Register und Segmente einsehen und natürlich auch den disassemblierten Code. Auch kann man Breakpoints setzen und so mit Run z.B. langwierige Schleifen abarbeiten. Den Turbo-Debugger starten Sie so: td Filename.ext . EXT = EXE oder COM.

4. Literatur:

Da es meiner Erfahrung nach schwierig ist Literatur zu finden (entweder vergriffen oder sauteuer) hier ein paar Tips (Diese Bücher waren auch meine Quellen):

1. Zuallererst nat. meines ;-) : Roming, Rhode. MITP. ISBN: 3-8266-0671-X Preis: 24,95 Eur 472 S.
2. Programmiersprache Assembler. Reiner Backer. ro ro ro. ISBN: 3-499-19249-7 Preis: 18.90 DM
3. Assembler Programmierung. Günter Born. Addison Wesley. ISBN: 3-89319-882-2 Preis: 39.90 DM
Es handelt sich hier um eine CD-ROM, mit dem Buch im PDF-Format, Reader dabei.
4. Assembler griffbereit. Joachim Erdweg. Vieweg. ISBN:3-528-05231-7 -->Nachschlagewerk
Es enthält die Prozessorbefehle bis 486'er und viele Int 21h Funktionen. Preis: 18 DM
5. Internet. So gibt es z.B. massig Vi.. Sourcecode. Sehr lehrreich !! Aber bitte auf die Kreationen aufpassen, ich möchte sie nicht auf meiner Platte haben.
6. Am besten zum Lernen von Assembler ist Sourcecode !
7. Assembler Programmierung von Wolfgang Link 8. Auflage. ISBN: 3-7723-8838-8.
Das Buch hat ca. 300 Seiten und beinhaltet eine CD. Preis: 79 DM
8. TASM. Sven Letzel, René Meyer. Internat. Thomson Publ. ISBN: 3-929821-83-4. Preis: 79 DM
Sehr gelungenes und umfangreiches Buch, anscheinend nicht mehr erhältlich wohl aber: MASM.

5. Nützliches:

1. Die .RADIX-Directive.

Normalerweise Interpretiert der Assembler alle Zahlen ohne Anhängsel als Dezimalzahl. Wer aber nicht dauernd h schreiben will kann dies durch .RADIX 16 ändern, alle Zahlen ohne Anhängsel werden dann Hexadezimal Interpretiert. Dezimalzahlen müssen dann mit einem d versehen werden ! .Radix 2 : alle Zahlen werden als Binärzahlen interpretiert.

2. Ein Hexeditor ist auch manchmal gut: [Hed](#) für Windows (204 KB), einer der Besten, mit Disassemblier-Funktion, für Menü einfach ESC drücken!

3. Ein [Glossar](#) in Englisch, sehr gut!

4. EditPlus. Toller Texteditor! Gibt es [hier](#). Und eine stx Datei speziell für asm-Dateien kann man hier [downloaden](#).

Rechtliches:

Diese Homepage ist **FREWARE** d.h. sie darf unentgeltlich aber **NUR KOMPLETT** und **UNVERÄNDERT** weitergegeben werden !

Ich übernehme **keinerlei Haftung** für durch diese Homepage entstandene eventuelle Schäden !

TASM, Turbo Assembler und Borland sind eingetragene Warenzeichen der Borland Corporation

Microsoft, Windows 95 und MASM sind eingetragene Warenzeichen der Microsoft Corporation

Bei **Leseproblemen** versuchen Sie Bitte: **min. 800x600 + Vollbild**

Der Autor, am 31.08.00

Bei Kritik und Anregungen: [Schreiben Sie mir !](#)

Neu: [PGP Key !](#)

Zuguterletzt: Noch ein paar [Links](#) !

[Zurück Weiter Inhalt](#)



Suchen: [Altavista](#), [Google](#), [Metafinder](#)

Homepage von [Joachim Rohde](#)! Viel zu Assembler, incl. übersetzte Tutorials!

The Art of Assembly Language [Programming!](#)

MASM32-Packet und Tutorials: [MOVSD](#)

Telepolis - Magazin von Heise, immer was interessantes: [Hier lesen!!](#)

Digital Mars C and C++ Compilers: [Digital Mars Free!](#)

Noch mehr: [Programmers Heaven - Assembler Zone](#)

Zuguterletzt: [Dolphinz Page](#)

Auch wenns angeblich nix bringt: Am 12. Mai 1998 hat das Landgericht Hamburg entschieden, dass man über einen Link die Inhalte der folgenden Seiten eventuell mit zu verantworten hat. Für alle Links zu fremden Seiten von meinen Seiten aus gilt: Ich habe keinen Einfluß auf die Inhalte der gelinkten Seiten. Deshalb distanziere ich mich hiermit ausdrücklich von allen Inhalten aller gelinkten Seiten und mache mir ihre Inhalte nicht zu eigen. Diese Erklärung gilt für alle auf meinen Seiten vorhandenen externen Links.

[Zurück Inhalt](#)



Hier noch etwas zum Inhalt unseres Buches (das weit über diese immer noch veraltete Homepage hinausgeht!) und am Ende noch ein Hinweis auf ein paar kleinere [Fehler der 1. Auflage](#) die sich in unser Buch eingeschlichen haben:

Inhaltsverzeichnis der 2. Auflage:

Inhaltsverzeichnis der **2. Auflage** gibt es [hier als PDF!](#)

Das Buch hat 600 Seiten und wird mit einer CD–Ausgeliefert. Neuigkeiten: Stark erweitertes Coprozessor Kapitel, neue Befehle, neue Aufgaben, Kapitel über den Befehl CPUID, Optimieren von Code, Reverse Engineering, Infos zu Linux und NASM.

Fehler in der 2. Auflage:

Bekanntermaßen ist es nicht möglich ein Buch ohne Fehler zu schreiben. Bis jetzt hab ich nur einen gefunden: Im Inhaltsverzeichnis steht NABFBFSM. Das soll wohl NASM heißen würde ich sagen. Ansonsten bis jetzt keine bekannt!

Kurze Inhaltsangabe (1. Auflage):

1. Theoretische Grundlagen: Bits, Bytes, Register, Stack usw.
2. Mathematisches: Hexadezimal, Dezimal, Dualsystem. Umwandlungen und Rechenoperationen. Halbaddierer usw.
3. Erste Programmierschritte: Assembler und Linker, "Hallo Welt" etc.
4. Möglichkeiten des Assemblers
5. Befehle und Adressierungsarten
6. DOS unter der Lupe: Interrupts, COM & EXE, PSP ...
7. Programmieren – Die Grundlagen: Arbeiten mit Texten und Zahlen, externe FAR–Prozeduren, Kommandozeile, Ports, Dateien
8. Grafikprogramme: VGA & Co, Feuer–Demo, Geschwindigkeitsoptimierung, Bildbetrachter
9. TSR's und Coprozessor.
10. Fehlersuche mit dem Debugger.
11. Assembler in Hochsprachen. Teil II des Buches (von Joachim): Assembler unter Windows.
12. Grundlagen zur Windows–Programmierung: Was ist ein API, eine DLL? MessageBox, 32–Bit, FLAT–Modell uvm.
13. Fenster, Dialogboxen und Ressourcen: Fenster, Handels, Nachrichten, Menüs, Steuerelemente, Standard–Dialoge...
14. Tools: Prostart, Libraries...
15. Debugging und Exception–Handling.
16. Prozesse–Threads und Timer
17. Dateien
18. DLL's
19. Registry
20. "Behind the scenes"
21. MMX und 3DNow!
22. Umfangreicher Anhang: Interruptfunktionen, Win32Api–Fktnen, Strukturen, Nachrichten und Lösungen der Übungsaufgaben.....

Inclusive CD–ROM! Insgesamt 474 Seiten!

Errata 1. Auflage:

1. **Feuer-Demo:** Hier ist ein Fehler im Macro "WaitRet": Darin sind die beiden bedingten Sprünge falsch: Richtig ist je und nicht jne, dann braucht das Programm unter schnelleren Rechnern auch nicht mehr gebremst zu werden (damals hatte ich nur recht langsame Rechner so daß das Problem nie wirklich deutlich rauskam). Nach ändern der Sprünge kann man also das Macro Delay und dessen Aufruf im Programm entfernen! Zudem ist es bei dem Programm nötig die Prozedur "Random" mit "Random ENDP" statt nur mit "ENDP" zu beenden, sonst streikt der MASM (nicht der TASM)! Hier das Programm, mit einem Zusatzfeature beim Beenden, und den Korrekturen zum [Download](#).
2. WinXP: Weder das Assembler-Programm Maus.com noch das Pascal Programm Maudemo.exe funktionieren unter WinXP! Auch bei anderen Programmen sind Einschränkungen möglich. Zudem ist es günstig, wenn man bei Programmen welche eine Ausgabe (Text,Zahl usw.) tätigen und sich dann selbst beenden, den Benutzer eine Taste drücken läßt. Sonst kann es beim Ausführen von Windows aus passieren, daß der Benutzer das Ergebnis nur extrem kurz zu sehen bekommt, da nach Programmende in der Regel auch das DOS-Fenster geschlossen wird. Dafür eignet sich Funktion 7 des int 21h kurz vor Programmende (also: mov ah,07h + int 21h vor mov ah,4Ch + int 21h bzw. vor ret oder int 20h).
3. Zur Frage "Welcher Assembler/Linker?" siehe [hier!](#)
4. Das 10,13 bzw. 13,10 am Ende eines Strings (also z.B. Hallo1 DB "Hallo!",10,13,"\$") bringt den Cursor an den Anfang der nächsten Zeile (ansonsten wird der nächste String direkt hinter dem vorherigen ausgegeben). Während unter DOS die Reihenfolge (also 10,13 oder 13,10) egal ist, so gilt unter Windows nur 13,10. Die Zahlen stehen für die Zeichen CR (carriage Return bzw. Wagenrücklauf, 13) und LF (line feed bzw. Zeilenvorschub, 10)
5. S. 94: Hier muß es heißen: "So speichert mov word ptr [bx],22h ab der Adresse **DS:BX** die Bytes [...] wohingegen mov byte ptr [bx],23h an der Stelle **DS:BX** [...]" also zweimal DS:BX statt DS:DX ganz am Anfang steht es noch richtig da.
6. S. 146: Soll natürlich 10 hoch -6 sein
7. S. 230: Hier Fehlt in der Zeile "myDialog" ein Komma am Ende sowie ein Hinweis, daß alles in eine Zeile muß!
8. S. 276: Die Texte von Parameter 4 und 5 sind vertauscht!
9. Folgende Befehle sollten eigentlich im Buch sein, wurden aber vergessen (sorry meine Schuld!). Zum Thema Vorzeichen siehe auch [hier das oberste Programm](#):

1. IDIV: Integerdivision (Vorzeichenbehaftet).

Syntax: IDIV Operand.

Der Befehl IDIV dividiert bei einem Byte-Operanden das AX-Register, bei einem Word-Operanden das Registerpaar DX:AX und bei einem Doubleword-Operanden (32-Bit) das Registerpaar EDX:EAX durch den Operanden. Der Operand kann ein Register oder eine Speichervariable sein. Bei einem Byte-Operanden geht das Ergebnis nach AL und der Rest nach AH. Bei einem Word- bzw. Doubleword-Operanden landet das Ergebnis in AX und der Rest in DX bzw. in EAX und EDX.

2. IMUL: Integerdivision (Vorzeichenbehaftet).

Syntax: IMUL Operand1 [,Operand2 [,Operand3]].

Der Befehl IMUL kann eine vorzeichenbehaftete Multiplikation mit a.) einem, b.) zwei oder c.) drei Operanden ausführen. Im Falle von a.) wird wie bei MUL vorgegangen. Bei b.) werden beide Operanden miteinander multipliziert und das Ergebnis wird im ersten Operanden abgelegt. Der erste Operand muß dabei ein 16- oder 32-Bit Register sein, der zweite Operand kann ein Register, eine Speichervariable oder eine Konstante sein. Bei c.) wird der zweite Operand mit dem dritten multipliziert und das Ergebnis im ersten Operanden abgelegt. Der erste Operand muß dabei ein 16- oder 32-Bit Register sein, der zweite Operand kann ein Register oder eine Speichervariable sein, der dritte Operand muß eine Konstante sein.

3. BSWAP: Byte Swap, little/big-Endian nach big/little-Endian.

Syntax: BSWAP Operand

Der Befehl BSWAP konvertiert von little-Endian nach big-Endian und vice versa. Der Operand ist ein 32-Bit Register.

4. Bedingte Sprünge:

JB: Sprünge wenn kleiner (below), geprüftes Flag: CF=1. **JNB:** Sprünge wenn nicht kleiner (not below), geprüftes Flag: CF=0. **JBE:** Sprünge wenn kleiner gleich (below equal), geprüfte Flags: CF=1 oder ZF=1. **JNBE:** Sprünge wenn nicht kleiner gleich (not below equal), geprüfte Flags: CF=0 und ZF=0.

5. WAIT / FWAIT:

Syntax: WAIT oder FWAIT

Wartet bis Coprozessor fertig ist.

Wird fortgesetzt...

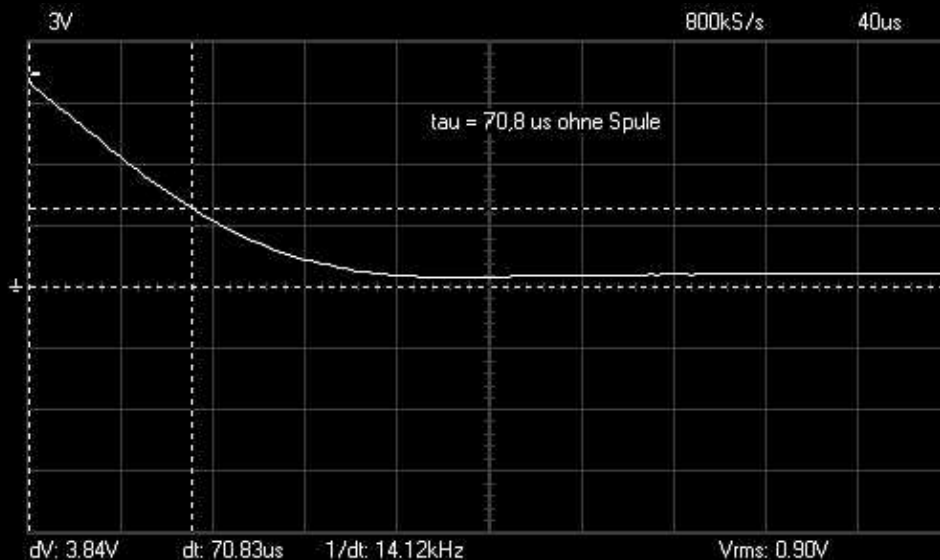
[Zurück!](#)




Hier werde ich noch zwei kleine Programme vorstellen:

Das Erste gibt eine *vorzeichenbehaftete 32-Bit Zahl* aus (Übergabe in EAX). Ich habe es als Ergänzung zu den Ausgabeprozeduren in unserem Buch programmiert. Sie bekommen es [hier!](#) (2 kByte)

Das Zweite ist von eher speziellem Interesse: Es kommuniziert mit der Oszilloskopsoftware der Velleman-Oszilloskope PCS100 und PCS500 (nur Kanal 1) über die DLL dsolink.dll. Es berechnet dabei unter anderem den Wert tau, also die Zeitkonstante bei einer Kondensatorentladung. Dabei sollte die aufgenommene Kurve für ein sinnvolles Ergebnis ungefähr (!) so aussehen:



Aufgenommen über Trigger an, Trigger Edge: "Arrow down" und Trigger Level knapp unter der Kondensatorspannung. Die Berechnung startet vom allerersten Punkt. Nähere Infos zur Nutzung der DSOLink.dll und somit des Programmes auch in der Hilfe des Oszilloskopes! Das Programm ist vor allem als Beispiel anzusehen... Keine Gewähr für die Richtigkeit der angezeigten Daten versteht sich. Das Win32-Assemblerprogramm gibt es mitsamt Quellcode [hier](#) (30 kByte). Angezeigt wird auch die Spitze-Spitze-Spannung sowie die Sample-Rate usw.:

Schnittstelle für DSO by Marcus Ro...

Datei

Lese CH1

Sample-Rate [Hz]: 8000000
Full Scale Voltage [mV]: 24000
Ground-Level [AD-Cnts]: 126

Spitze-Spitze [AD-Cnts]: 100
Spitze-Spitze [mV]: 9375
Tau, gerundet [Mikrosec.]: 71

[Zurück!](#)



Laden Sie sich das MASM32 Packet hier: <http://www.codingcrew.de/masm32/index.php> (deutsche Seite) oder hier: <http://www.masm32.com/masmdl.htm> (englische Seite).

Das Packet bietet weit mehr als nur Assembler und Linker!

Die Vorgängerversion 7.0 finden Sie hier: <http://win32asm.cjb.net/> unter "Downloads". Für DOS wird ein anderer Linker benötigt:

- Der Linker des MASM32 Packetes ist ein 32 Bit Linker für Windows-Projekte. Für DOS benötigen Sie einen anderen (16-Bit) und zwar von hier: <ftp://ftp.microsoft.com/softlib/mslfiles/link563.exe> Falls Sie mit Ihrem Browser keinen Erfolg haben müssen Sie ein FTP Programm nehmen oder den Linker bei <http://win32asm.cjb.net/> herunterladen (ebenfalls unter Downloads)
- Führen Sie lnk563.exe aus um link.exe zu erhalten, befördern sie diese dann nach \masm32\BIN\ (Vorher umbenennen um den Linker parallel zum 32 Bit-Linker verwenden zu können). Fügen Sie diesen Pfad auch in ihrer Path-Variable in Autoexec.bat ein. Die beiden anderen Dateien aus lnk536.exe können Sie wieder löschen!

Syntax für Masm: Bei EXE: ML /c filename.asm sowie link filename.obj
und bei COM: ML /c /AT filename.asm sowie link /t Dateiname.obj

Das /c verhindert jeweils automatisches Linken (produziert meist Müll). ".asm" und ".obj" wird nicht automatisch angehängt! Wenn sie den DOS-Linker umbenannt haben müssen die Eingab natürlich entsprechend modifizieren!

[Zurück Inhalt](#)



[01. Einleitung](#)

[02. Der 8086'er Prozessor](#)

[03. Zahlensysteme](#)

[04. Das erste Programm](#)

[05. Das erste Programm erklärt.](#)

[06. Der Aufbau eines Assemblerprogramms.](#)

[07. Wichtige 8086'er Befehle \(bis mul\).](#)

[08. Wichtige 8086'er Befehle \(bis xor\).](#)

[09. Programm "Wie geht's".](#)

[10. Programm "Num On/Off".](#)

[11. Programm "CMOS-Tools".](#)

[12. Programm "Konvert".](#)

[13. Bisherige Interruptfunktionen.](#)

[14. Programm "CRC 16".](#)

[15. Anhang](#)

[16. Links](#)

[AA. Kostenloser Assembler](#)

★